Safety-critical design of the generic driving actuator
a hybrid approach

Merkx, L.L.F.

Award date:
2008

Eindhoven University of Technology

Department of Computer Science
Section Design and Analysis of Systems

Safety-critical design of the Generic Driving Actuator

A hybrid approach

L.L.F. Merkx

Master's thesis

Exam committee:

Chairman	Prof. dr. ir. J.F. Groote
Mentor	Dr. ir. P.J.L Cuijpers
External member	Dr. ir. R.J. Bril
Mentor (TNO)	Ir. H.M. Duringhof

Exam date:
November 20, 2007

Summary

At TNO Automotive the Generic Driving Actuator (GDA) is developed. The GDA is a device capable of driving a vehicle fully automatically using the same interface as a human driver does. In this thesis, the design of the GDA is discussed and the interaction between its software and hardware components is analyzed from a safety point of view.

Before starting with the safety issues of the GDA, a control strategy is designed for the GDA. A torque controller and a position controller is built and their performance is analyzed. The torque controller is a feedforward controller without feedback and the position controller is implemented as a PID controller. These fairly simple controllers perform good enough for their use in the GDA.

The hardware design of the GDA is adapted to achieve an acceptable level of fault tolerance. Safety must be guaranteed for single-point failures. To achieve this, safety-critical hardware components are duplicated without making the system less compact or too costly.

The software of the GDA needs to be correct under all circumstances to guarantee its safety. Therefore the software is designed and verified using formal methods. First the requirements of the software are specified. Based on these requirements a software model is built in the proces-algebraic language μCRL. The requirements are then converted to modal logic, which can be verified against the software model.

Finally, a simulation model is built in Simulink to analyze safety during driving tests in simulation. Furthermore a strategy is designed to maintain safety during component failures and external emergencies. The simulation model consists of a vehicle model and a model of the GDA with functionality to simulate faults of the GDA.

Preface

This thesis is written at TNO Automotive for the department of *Integrated Safety* as part of a joint thesis for both Computer Science and Mechanical Engineering. This thesis focusses on the computer science aspects to obtain the master degree in computer science, while the other thesis focusses on the mechanical engineering aspects. The joint thesis is supervised by prof. dr. Henk Nijmeijer (TU/e Mechanical Engineering), prof. dr. ir. Jan-Friso Groote and dr. ir. Pieter Cuijpers (TU/e Computer Science) and ir. Hans-Martin Duringhof (TNO Automotive).

This thesis is written at TNO Automotive for the department of *Integrated Safety* as a joint thesis for both Computer Science and Mechanical Engineering to obtain the master degree. This master's thesis is supervised by prof. dr. ir. Jan-Friso Groote and dr. ir. Pieter Cuijpers (TU/e Computer Science), prof. dr. Henk Nijmeijer (TU/e Mechanical Engineering) and ir. Hans-Martin Duringhof (TNO Automotive).

During this master's thesis, I had a lot of support from my supervisors. I thank them for giving me the opportunity to make a joint thesis despite of the complications this would bring with it. Fortunately they liked the challenge.

In addition to the support from my supervisors I received advice and assistance from ing. Fred Gordebeke and ing. Joris Coolen (electrical implementation of GDA, TNO), ir. Robert Verschuren (Steering robot, TNO), ir. Roel Leenen and ir. Raymond Tinsel (MACS/Simulink, TNO), ir. Pieter Schutyser and ir. Arjan Teerhuis (Simulink, TNO), dr. ir. Antoine Schmeitz and dr. ir. Igo Besselink (vehicle dynamics, TNO / TU/e), ir. Muck v.d. Weerdenburg and dr. ir. Wieger Wesselink (μCRL, TU/e), ir. Erwin de Groot (Review of report, UCalgary) and others.

Furthermore I want to thank the people who made this combined study possible in the first place. In 2001 I wanted some variation in my computer science curriculum. I wanted to do some automotive classes and ultimately prof. ir. Nort Liebrand convinced me of doing both curricula: computer science and mechanical engineering. He arranged my individual curriculum for mechanical engineering together with dr. ir. Bram de Kraker (director of education) and ir. Elise Quant. Dr. Ad Aerts (student advisor) helped them finding overlap between the two curricula and adjusted the computer science curriculum.

Last but certainly not least, I like to thank my friends, my family and my girlfriend for their support throughout this master's thesis.

Eindhoven, November, 2007

Leon

Nomenclature

Latin symbols

Symbol	*Description*	*Unit*
d_{column}	Damping constant of steering column	[Nms/Rad]
d_{gda}	Damping constant of the steering robot	[Nms/Rad]
d_{rack}	Damping constant of the steering rack	[Ns/m]
f_{act}	Actuator frequency	[Hz]
f_s	Sample frequency	[Hz]
i_{st}	Steering ratio	[−]
k_{column}	Spring constant of the steering column	[Nm/Rad]
k_{gda}	Spring constant of the steering robot	[Nm/Rad]
l	Beam length	[m]
m_{rack}	Mass of the steering rack	[kg]
n_p	Number of pole pairs	[−]
n_{rack}	Ratio between steering rack and steering column	[m/Rad]
t_{delay}	Delay time	[s]
t_s	Sample time	[s]
x_{rack}	Position of the steering rack	[m]
$C(s)$	Controller transfer function	
F	Force	[N]
F_y	Lateral tyre force	[N]
F_z	Vertical tyre force	[N]
F_{wh}	Wheel forces and torques (acting on vehicle body)	[N, N, N, Nm, Nm, Nm]
I_d, I_q	Currents in rotor frame	[A]
I_u, I_v, I_w	Currents in three-phase stator frame	[A]
I_x, I_y	Currents in xy-frame	[A]
J_{gda}	Inertia of steering robot	[Nms^2/Rad]
K	Controller gain	[−]
K_{gda}	Friction coefficient of the steering robot	[Nm]
K_{rack}	Friction coefficient of the steering rack	[N]
K_t	Torque constant	[Nm/A]
K_u	Ultimate gain	[−]
M_x	Self-aligning moment of the tyre	[Nm]
T_d	Differential control action	[s]
T_d	Damping torque	[Nm]
T_{eff}	Effective torque	[%]
T_{fric}	Friction torque	[Nm]

T_{gda}	Torque applied by steering robot	[Nm]
T_i	Integral control action	[s]
T_k	Spring torque	[Nm]
T_m	Motor torque	[Nm]
T_{st}	Steering torque	[Nm]
T_u	Ultimate period	[s]

a	Action variable	
b	Boolean variable	
p	Process variable	
q	Process variable	
t	Time	[s]
Y	Propositional variable	

Greek symbols

Symbol	*Description*	*Unit*
α	Angle	[Rad]
$\alpha_{backlash}$	Backlash in steering robot	[Rad]
α_m	Angle of pole	[Rad]
α_{st}	Steering wheel angle	[Rad]
α_{wh}	Steer angle of the wheel	[Rad]
γ	Electrical angle	[Rad]
λ	Failure rate	[s^{-1}]
ψ_d, ψ_q	Rotor fluxes	[T]
Δ	Difference	

α	Action formula	
β	Regular formula	
δ	Inaction	
μ	Least fixed point operator	
ν	Greatest fixed point operator	
τ	Internal action	
ϕ	State formula	

Other symbols

Symbol	*Description*	*Unit*
$\top$	Set of all actions	

Operators

Symbol	*Description*
$+$	Non-deterministic choice
$\cdot$	Sequential composition
$\|\|$	Parallel composition
$\lhd\ \rhd$	Conditional choice
$\neg$	Logical negation operator
$\vee$	Logical or operator
$\wedge$	Logical and operator
$\cdot$	Concatenation operator
$\|$	Choice operator
*	Transitive-reflexive closure
$^+$	Transitive closure

Abbreviations

μCRL	micro Common Representation Language
ACP	Algebra of Communicating Processes
CADP	Cæsar Aldébaran Development Package
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
GDA	Generic Driving Actuator
HML	Hennessy-Milner Logic
HyPA	Hybrid Process Algebra
ISO	International Organization for Standardization
LPO	Linear Process Operator
MACS	Modular Automotive Control System
PID	Proportional-Integral-Differenial (controller)
PLTL	Propositional Linear-Time Logic
SAE	Society of Automotive Engineers

Contents

Chapter 1

Introduction

'Computer Science is no more about computers than astronomy is about telescopes.'

Edsger W. Dijkstra

In this chapter an introduction is given to the master's thesis. The project background, the project description and the structure of the thesis are described.

1.1 Background

TNO Automotive is an independent research institute for the automotive industry. In Helmond the department of *Integrated Safety* is situated. One of their research areas is the development and testing of intelligent vehicle systems.

A new project in this area is the development of the so-called Generic Driving Actuator (GDA). The GDA is a compact system which can drive a vehicle fully automatically. This system uses the same interfaces as a human driver does: it controls the steering wheel, the brake pedal and the throttle pedal. The GDA consists of three robots to control each interface: a steering robot and two pedal robots. The clutch pedal will be ignored, since most vehicles also have a version with automatic gearbox, which can be used instead.

The main use of the GDA will be to perform driving tests. The GDA can be a good alternative for a test driver. One can think of driving tests which would endanger a test driver (roll-over tests) and driving tests which are hard to (re)produce (sinusoid steer and step steer tests). These driving tests can be used for system identification[1] or vehicle dynamics assessment.

Although the control system's basic functionality will be based on fairly simple feedback control techniques (e.g. PID control), special attention should be paid

[1] System identification is a general term to describe mathematical tools and algorithms that build dynamical models from measured data. [WIK]

to safety. Failing control can lead to life threatening situations, when the GDA is used. One can imagine that simply shutting down the GDA is not always the wisest decision.

When the master's thesis was started, the design of the GDA was already developed (without safety measures) and the hardware components were finished and available. The design had not yet been tested in reality.

1.2 Project description

In this master's thesis the design of the GDA is described. The objective of the thesis will be on how to make the GDA a safe system. A basic control system (single actuator, single controller and single sensor) is built and will be improved by applying safety strategies. The safety will be evaluated from three different perspectives:

- *Hardware design*: Fault-tolerant strategies are applied to reduce the risk of failing hardware.
- *Software design*: Formal methods are used to verify the correctness of the software.
- *Vehicle dynamics*: Research is done on the effect of failing GDA hardware on vehicle dynamics and how to continue safely in case of failing hardware or an emergency (e.g. another vehicle which accidently obstructs the driving test).

The subject of this master's thesis will be:
The safety-critical design of the Generic Driving Actuator

During the thesis a lot of problems with the original design of the steering robot were detected. Getting the steering robot running, which was supposed to be an easy job, turned out to be a laborious, costly and time-consuming task. The original design principles did not seem to work and components had quirks or did not deliver the required specification at all.

The steering robot has only worked with poor performance. After many improvements to the design and to the components the steering robot is almost finished, but still not working optimally.

The work on the pedal robot has been delegated to others due to time constraints. Although the GDA is now close to completion, the project has been postponed due to marketing problems, new strategic plans and reorganization.

Without having the full GDA operational, tests in reality are not feasible. This means that the development of a safe GDA can only be done with simulations, without being able to verify the results in reality.

1.3 Structure of report

In this thesis, first the system requirements on the Generic Driving Actuator are introduced in chapter 2. Focussing on the needs of the different driving tests and the competition, the specifications for the GDA are described and the importance of safety is stressed.

In chapter 3 the controller of the GDA is designed without any safety measures. Since the three robots have a similar hardware design, only the steering robot design is described in this chapter. First the hardware components are discussed followed by the development of the torque controller and the position controller.

Chapter 4 describes the safety-critical hardware design. The safety of the GDA must be guaranteed for single-point failures. To achieve an acceptable level of fault tolerance the safety-critical hardware components need to be duplicated. Some safety theory will be presented and put into practice.

Chapter 5 describes the software design and verification of the control system. Formal methods will be used to guarantee the requirements of the GDA's software. An introduction is given on safety-critical software design and how formal methods fit in. Formal methods are used to specify, design and verify the GDA's software. In the last section the implementation is described in Simulink.

In chapter 6 the safety during driving tests is analyzed in simulation for both hardware and software. A strategy is designed, what should be done to maintain safety during component failures. A simulation model will be built with the simulation platform *Simulink*. Only the behaviour of the steering robot will be designed in complete detail. The other robots will be simplified in simulation. A practical approach to validate the models by system identification experiments will be proposed.

Chapter 7 contains the conclusions and further recommendations.

A short discussion is given in chapter 8 on the differences between computer science and mechanical engineering and how these fields should be combined. As the master's thesis also will show, combining both fields will improve the design proces and remove problems in an early phase by verifying requirements.

In figure 1.1 the structure of the report is schematically shown. The relations between different parts of the report are indicated and it shows to which field these parts belong. This master's thesis is written for both mechanical engineering and computer science and therefore certain parts will not immediately be clear for people who are not familiar with both fields.

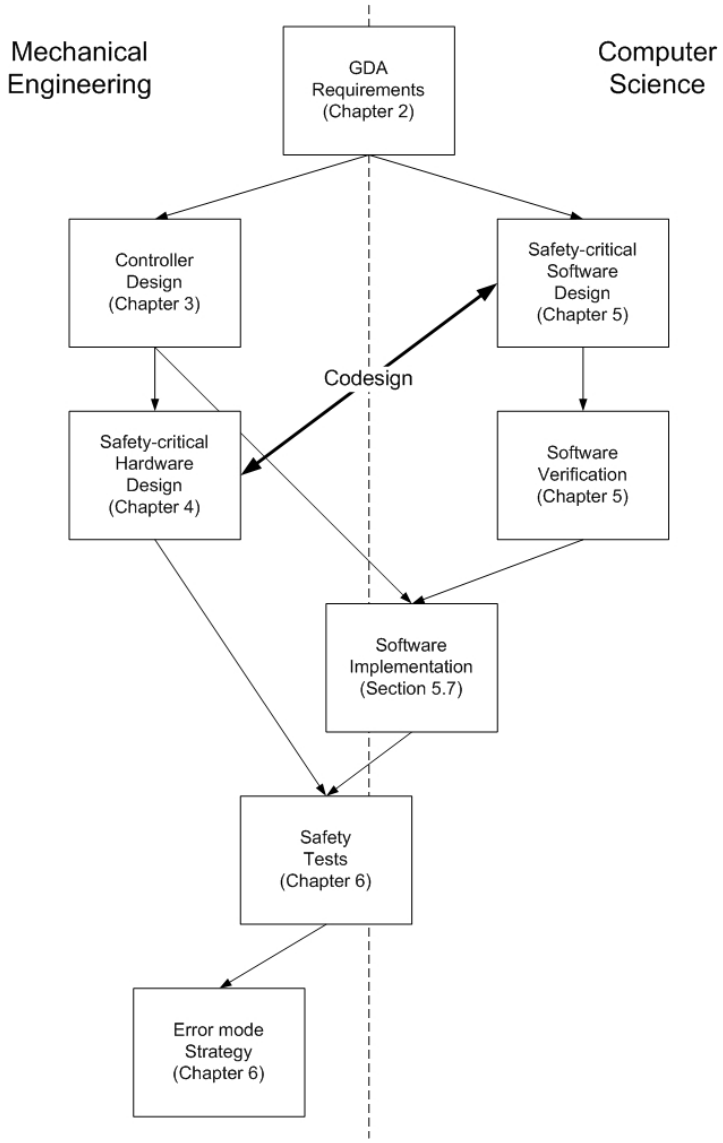

Figure 1.1: Structure of report

Chapter 2

Generic Driving Actuator

Three laws of robotics
1. A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Isaac Asimov

This chapter introduces the GDA. First a general introduction is given and the importance of safety is discussed. The last section contains the specifications of the GDA, based on the needs of the different driving tests and the competition.

2.1 Introduction

The GDA is a device which is capable of driving a vehicle fully automatically using the same interfaces as a human driver does. It is able to control steering wheel, braking pedal and throttle pedal. The clutch pedal will be ignored, since most vehicles also have a version with automatic gearbox, which can be used instead.

The GDA consists of a control system and three robots each with their own responsibility: turning the steering wheel, pressing the brake pedal and pressing the throttle pedal[1]. The control system receives setpoints from the user like steering wheel angle. The GDA design is schematically shown in figure 2.1. Note that all three robots make use of the control system, since the controllers of the three robots are also part of the control system.

[1]Modern vehicles more and more use throttle-by-wire. The traditional throttle cable is replaced by electronics. This makes it possible to use the electronic interface, instead of using a pedal robot for throttle control.

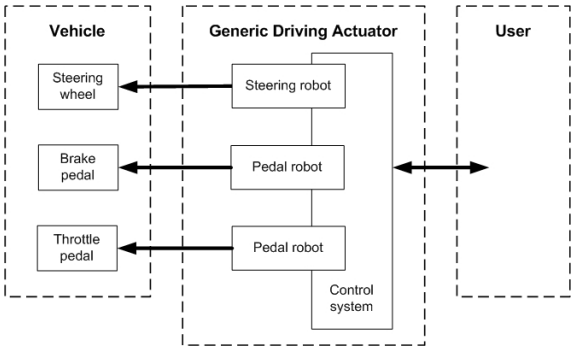

Figure 2.1: GDA Design

Two different types of robots are used in the design of the GDA: a steering robot and a pedal robot. Both types have an almost identical setup. The only difference is the output, which is either rotational (steering) or translational (pedal). The setup of the steering robot will be further explained in chapter 3. The pedal robot uses a similar setup with an additional component (a spindle) to convert the rotating motion into a translating motion. The steering robot is shown in figure 2.2 and the pedal robot is shown in figure 2.3.

Figure 2.2: Steering robot installed in Smart

The main use of the GDA will be to perform driving tests for system identification and vehicle dynamics assessment. The GDA can replace the test driver in dangerous driving tests (roll-over tests) and driving tests which are hard to (re)produce (sinusoid steer and step steer tests). Driving tests will be further explained in section 2.3.

It is also possible to use the GDA to test intelligent vehicle systems. An example of an intelligent vehicle system is *Automated Vehicle Guidance*, which let a vehicle follow a predefined trajectory. To test this system the steering, braking

Figure 2.3: Pedal robot

and throttle mechanism of the vehicle need to be adapted. Instead of converting the complete vehicle, the GDA can be used and the system can be tested directly. This will reduce development time and cost significantly.

2.2 Safety

The safety of the GDA is one of the most important design criteria. If a vehicle gets out of control, it poses a serious thread to the test driver and to the environment. This potential thread needs to be minimized.

If the GDA is used in combination with a test driver, he can take over control by shutting down the GDA in emergency situations. The steering robot and pedal robot are designed in such a way, that it is still possible for a test driver to drive the vehicle. However the test driver is not always fast enough to prevent dangerous situations. For example if the GDA decides to turn left, due to a failing hardware component, the test driver might not have a chance to respond on time to prevent a catastrophe. If the GDA is used for autonomous driving, the safety issues are even more serious.

The GDA can be considered to be sufficiently safe, if these three requirements are fulfilled.

- *A single-point-of-failure in hardware does not effect the proper operation*
  If one hardware component fails, it must still be possible to use the GDA to control the vehicle. Dangerous situations due to a single component failure are avoided this way. The hardware safety is described in chapter 4.

- *The software is safe*
  The software requirements must be verified to guarantee the correct behaviour under all circumstances. The software safety is further explained in chapter 5.

- *The GDA must be able to stop the vehicle safely by itself*
  If a hardware component fails or if an emergency occurs, the GDA must be able to stop the vehicle safely fully automatically. How this can be done is described in chapter 6.

2.3 Driving tests

The GDA is used for driving tests. There are many driving tests defined within the automotive industry to test handling, ride comfort, etc. In this section the most important driving tests for horizontal dynamics are discussed shortly.

2.3.1 Control response tests

Control response tests are driving tests to measure the response of a vehicle to inputs from steering wheel, throttle and brakes. These tests can be used to identify vehicle parameters.

- *Steady state cornering*
  These driving tests (also known as *Circle tests*) are described in [SAE96] and are performed to measure steady state response to steer input at different speeds. The most important reason to do these kinds of tests is to investigate the understeer/oversteer behaviour.
- *Lateral transient response tests*
  These driving tests are described in [ISO03] and are used to analyze the dynamic behaviour of a vehicle. This analysis is done in the time domain and in the frequency domain.

  The driving tests in the time domain are usually only used to judge the dynamic response of a vehicle subjectively. Large overshoots in yaw velocity, roll angle and sideslip angle during these tests are not accepted. The driving tests in the time domain are the following:

  - *Step input*: A steering wheel angle is set almost instantly (at speeds of around 1000 degrees/s) and kept at this angle at a specific vehicle speed.
  - *Single-cycle sinusoid*: In this test the steering wheel is rotated in a single-cycle sine motion at a specific throttle position. This test is similar to the lane-change test, which is described later.

  With the driving tests in the frequency domain, transfer functions can be computed. The driving tests in the frequency domain are the following:

  - *Random input*: Continuous pseudo-random input is applied to the steering wheel with a frequency of 0.2 Hz to 2 Hz at a specific vehicle speed.
  - *Pulse input*: A triangular waveform is applied to the steering wheel at a specific throttle position.
  - *Continuous sinusoidal input*: A sinusoidal input is applied to the steering wheel with a frequency starting at 0.2 Hz and ending at 2 Hz at a specific vehicle speed. This frequency is slowly and stepwise increased.

- *Brake and throttle tests*
  So far, the described driving tests only concerned the effect of steering. Other driving tests were developed to test the response on brake and throttle pedal. This can be done purely longitudinal, i.e. braking and accelerating in a straight line. However driving tests investigating the combined effect of steering and accelerating/decelerating are most interesting. One can think of the following driving tests: *Braking in a turn*, *Power-off in a turn* (throttle off) and *Power-on in a turn* (full throttle).

The described driving tests will be performed turning both left and right. The behavior of the vehicle turning left or right may not be symmetrical, since the vehicle itself will never be completely symmetrical. Therefore it is important to investigate this as well.

2.3.2 Handling tests

Handling tests are driving tests to measure the behaviour of the vehicle in combination with the driver, i.e. in handling tests the driver is part of the control loop. For this reason these tests are less objective than the control response tests. Handling tests are performed to see which car behaves best during particular maneuvers and to see if the vehicle's behaviour is acceptable in limit situations. These maneuvers are based on events occurring in real life.

The most important handling tests are described below:

- *Double lane change*: The test driver changes lanes twice and tries to drive the prescribed trajectory as fast as possible. The driving test trajectory is described in [ISO99].
- *Elk*: This driving test is similar to the Double lane change test, only it has somewhat different specifications.
- *J-turn*: This driving test simulates the scenario in which a driver steers away from an obstacle. A step function is applied to the steering wheel at the highest possible speed. This test is developed to test rollover propensity.
- *Fish hook*: This driving test simulates the scenario in which a driver first steers to one direction and than overcompensates to the other direction. This test drive is run at the highest possible speed. The test is also developed to test rollover propensity.

2.4 Competitors

There are some competitors in the same market building systems comparable to the GDA. However these companies can only build components (a steering robot and/or a pedal robot) or build robots for tests on test stands. None of the competitors offers a compact, lightweight driving robot to perform driving tests, which uses the onboard 12V charge. It does not need additional batteries and/or transformers. Furthermore it is quick to install in a random vehicle without adjustments to the vehicle.

2.4.1 History

Until recently R&D departments of car manufacturers built their own applications to perform driving tests. These were electro hydraulic, making them large, heavy, complex and expensive. Therefore it used to be a laborious task to build the driving robot into a vehicle. It was also impossible to drive the car manually when such a robot was installed. If the vehicle can be driven manually, the driving test is easier to perform. In this way the driving test can easily be started, when the vehicle reaches the initial speed on a specific part of the test track. However it was possible to perform driving tests.

Only since recently it is possible to make compact, light-weight steering and pedal robots with modern electro mechanics. The market for these applications is therefore relatively new and small.

2.4.2 Steering robot

Modern steering robots are capable of performing standard driving tests. They are relatively easy to install in modern vehicles and it is possible to drive the vehicle manually with the equipment installed.

The most important competitor is Anthony Best Dynamics (ABD). This company builds two different steering robots: a steering robot for regular driving tests and a steering robot for parking tests. The steering robot is considered to be the best in the market and is used by many companies.

Other steering robot vendors are ATI/Heitz and Stähle. The main difference between these vendors and ABD is the user interface. The ABD steering robot is much easier to operate. It has a touch screen to select and adjust driving tests and to show the results real time. Furthermore it is possible to correct the steering robot by joystick (e.g. if the car leaves the track). In figure 2.4 the ABD robot [ABD] and the ATI/Heitz robot [ATI] are shown.

Figure 2.4: Steering robots: ABD (left) and ATI/Heitz (right)

The steering robot specifications are compared in table 2.1.

	Maximum torque (Nm)	*Maximum speed (deg/s)*
ABD SR30 [ABD04]	18 (nominal) 33 (peak)	1900 (at zero load) 1440 (at 10 Nm) 425 (at peak load)
ABD SR30 High Power System [ABD04]	30 (nominal) 33 (peak)	2350 (at 7 Nm) 1440 (at 22.5 Nm) 850 (at peak load)
ATI/Heitz Sprint 3 [ATI97]	23 (nominal) 50 (peak)	1300 (at peak load)
Stähle SSP2000 [STA]	50 (nominal)	1800 (at unknown load)

Table 2.1: Steering robot specification of competitors

2.4.3 Pedal robot

The most important developers of pedal robots are ABD and Stähle. In figure 2.5 the ABD robot [ABD] and the Stähle robot [STA] are shown.

Figure 2.5: Pedal robots: ABD (left) and Stähle (right)

The Stähle Autopilot can control all pedals and can drive the vehicle automatically. It is developed for use on test stands to do for example exhaust emission tests or durability tests. It is not the perfect solution for driving tests, because it is not possible to drive the vehicle manually anymore without removing the system. Furthermore it is almost impossible to install a steering robot due to its size.

Stähle also has a robot for individual pedals (AP-GB/2.15). But still, for the same reasons, it is not really suitable for performing driving tests

ABD has only built a braking robot. With this robot it is still possible to operate the brake pedal, when the braking robot is not in operation. It is easy to install and it is possible to do a large variety of tests (step, pulse, sinusoid, etc). This makes it a perfect robot for driving tests.

A third company, Horiba Instruments, also makes pedal robots. These robots

are similar to the Stähle Autopilot.

The pedal robot specifications are compared in table 2.2.

	Maximum force (N)	*Maximum speed (m/s)*	*Travel (m)*
ABD BR1000 [ABD]	1400	0.80 (at zero load) 0.70 (at 400 N)	-
ABD BR2000 [ABD]	2100	0.55 (at zero load) 0.49 (at 800 N)	-
Stähle AP-GB/2.15 [STA]	350	0.30 (at unknown load)	0.15

Table 2.2: Pedal robot specification of competitors

2.5 Specification

The specification of the two robots to be built is based on the needs of the driving tests and it needs to be comparable to the competition.

2.5.1 Steering robot

The maximum steering efforts of a human driver are measured first. Low speed tests (5 km/h) with a BMW 3-series showed that the maximum rotational speeds where around 720 degrees per second. The highest torques measured were around 10 Nm during hard cornering. During normal driving speeds, the torques will almost never exceed 5 Nm.

The driving tests require rotational speeds of up to 1000 degrees per second. However if rotational speeds are increasing (e.g. during driving tests), the steering wheel torques are rapidly increasing as well. This is due to the fact that the power steering mechanism can no longer power the steering mechanism. It can even give resistance to the steering wheel torque instead.

The required nominal torque is set at 25 Nm and it needs to be available at 1000 degrees per second, when the torque is needed most. The GDA specification is comparable with the competition.

2.5.2 Pedal robot

European legislation prescribes that the travel distance of a braking pedal is at most 15 cm [AMN86]. Furthermore it prescribes that the braking system must withstand pedal forces of 1500 N. This force is related to the highest possible force a human can apply.

TNO estimated the pedal forces and speeds to be similar to the ABD BR2000 (see table 2.2) during actual driving tests. In conclusion, there is no need to produce more than 1500 N and at 800 N brake pedal speeds must be reached of at least 0.5 m/s by the GDA.

Chapter 3

Controller design

'C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.'

Bjarne Stroustrup (Inventor C++)

In this chapter the design of the GDA controller without safety measures is discussed. Since the three robots have a similar hardware design, only the steering robot design is described in this chapter. First the hardware components are discussed and in the next sections the torque controller and the position controller are developed respectively.

3.1 Hardware

The steering robot has an electromechanical actuator installed, which is used to turn the steering wheel. The control system performs torque and motion control and an extra component converts controller output to actuator input. The steering robot also has two types of sensors installed: an angle sensor to give motion feedback and a torque sensor for test analysis purposes.

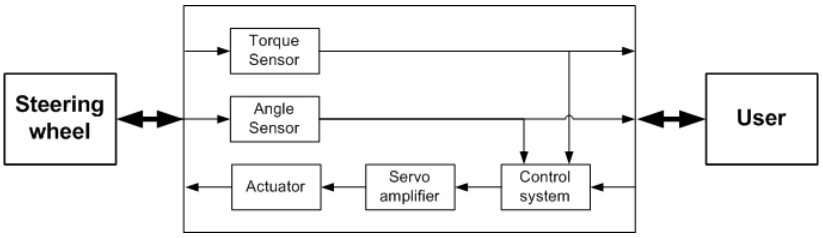

Figure 3.1: Hardware setup of the steering robot

The steering robot is schematically shown in figure 3.1 and the components are described below.

- *Actuator*: The steering wheel is driven by a brushless AC motor. Brushless AC motors are used, because DC motors suffer from too much torque ripple due to the abrupt voltage changes. The actuators drive the main axle via a drive belt.
- *Servo amplifier*: A brushless AC motor uses three phase sinusoidal commutation[1], which is fed by a servo amplifier.
- *Angle sensor*: The absolute steering wheel angle is sensed by a Netzer rotational electric encoder (see appendix A.1).
- *Torque sensor*: The steering wheel torque is sensed by strain gauges (see appendix A.2).
- *Control system*: The MACS (Modular Automotive Control System) will be used to compute the system's control actions. The MACS is a rapid control prototyping tool developed at TNO. With this tool it is possible to design real-time controllers for dynamical systems in a fast and efficient way. It will be used to control steering torque and steering wheel angle, which will be described in the following sections.

3.2 Torque controller

In the first section the design of the torque controller is described. In the second section the performance is analyzed.

3.2.1 Design

The used setup is schematically shown in figure 3.2.

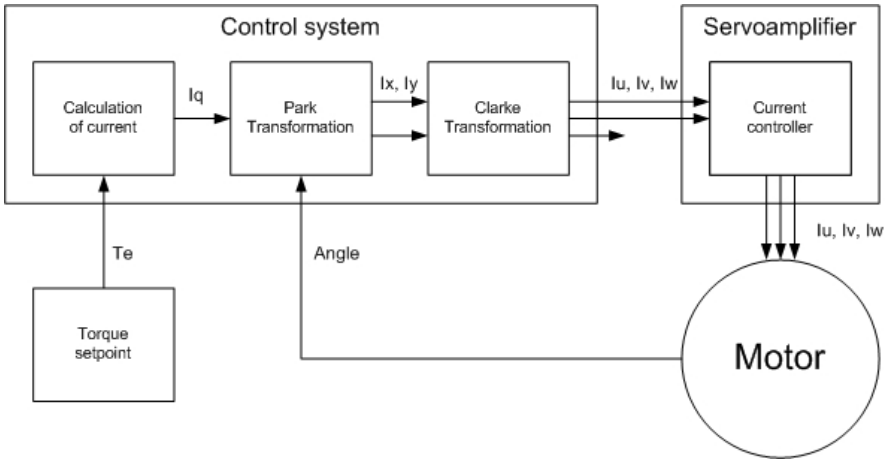

Figure 3.2: Schematic setup of torque control

[1]Commutation is the action of applying currents or voltages to the proper electrical motor phases so as to produce optimum motor torque at a motor's shaft [WIK]

From the required torque, the required currents in the rotor can be computed. These rotor currents need to be converted to currents in the stator coils. This conversion between rotor and stator frame is first explained.

In figure 3.3 an AC motor is shown with one pole. The rotor position is given in the dq-frame and the position of the stator coils is given in the uvw-frame.

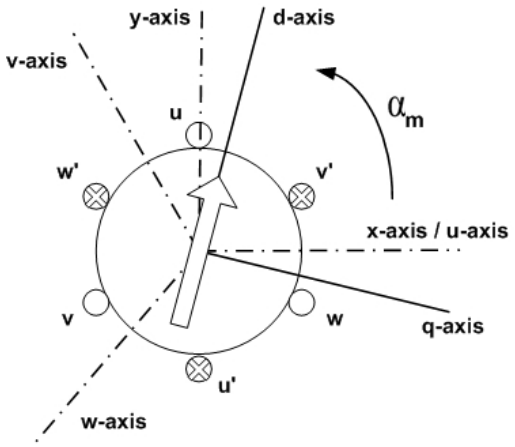

Figure 3.3: Different coordinate systems of the AC motor

The currents in the rotating dq-frame can be converted in the static xy-frame by using the angle of the pole (α_m). This transformation is called the *Park transformation* [DV04].

$$\begin{bmatrix} I_x \\ I_y \end{bmatrix} = \begin{bmatrix} \cos\alpha_m & -\sin\alpha_m \\ \sin\alpha_m & \cos\alpha_m \end{bmatrix} \cdot \begin{bmatrix} I_d \\ I_q \end{bmatrix} \tag{3.1}$$

The currents in the two-phase frame can be converted in the three-phase frame. This transformation is called the *Clarke transformation* [DV04].

$$\begin{bmatrix} I_u \\ I_v \\ I_w \end{bmatrix} = \sqrt{\frac{2}{3}} \cdot \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \cdot \begin{bmatrix} I_x \\ I_y \end{bmatrix} \tag{3.2}$$

The motor torque (T_m) can be computed in the dq-frame from the rotor fluxes (ψ_d and ψ_q), the rotor currents and the number of poles (n_p) according to [DV04].

$$T_m = \frac{3}{2} n_p \left(\psi_d \cdot I_q - \psi_q \cdot I_d\right) \tag{3.3}$$

The previous equation can be simplified, since there is no flux in the q-direction $\psi_q = 0$. The current I_d must be kept zero, otherwise the rotor magnet will be demagnetized and this current will not contribute to the motor torque.

$$T_m = \frac{3}{2} n_p \left(\psi_d \cdot I_q\right) \tag{3.4}$$

The motor torque is also given by the following equation.

$$T_m = K_t \cdot I_q \text{ where } K_t\text{: Torque constant} \tag{3.5}$$

The steering torque (T_{st}) is the product between the steering ratio (i_{st}) and motor torque (T_m).

$$T_{st} = i_{st} \cdot T_m \tag{3.6}$$

The steering torque can now be converted to rotor currents by combining (3.5) and (3.6).

$$T_{st} = i_{st} \cdot K_t \cdot I_q \tag{3.7}$$

This steering torque can then be converted to stator currents by using (3.1) and (3.2) to obtain the phase currents needed for the servo amplifier. The resulting Simulink block is shown in figure 3.4. In this model only two out of three stator currents are outputted. The missing third phase is internally computed by the servo amplifier.

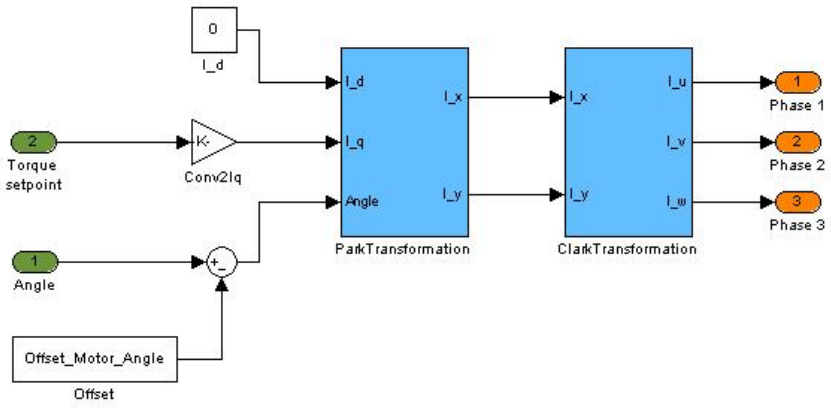

Figure 3.4: Torque controller block in Simulink

The offset between measured angle and motor angle is estimated. The offset is used, which has the largest rotation speed or highest torque at a specific current setpoint.

The torque constant (K_t) also needs to be computed experimentally. This calibration needs to be done, since a difference is to be expected between the theoretical and the delivered torque. This is primarily due to differences in the actual setup and the setup used to retrieve the motor specification.

The calibration procedure is shown in the figure 3.5. A small metal beam is attached to the steering robot's axle and a force transducer is attached to the beam to measure force in steady state. The delivered torque can easily be computed with (3.8).

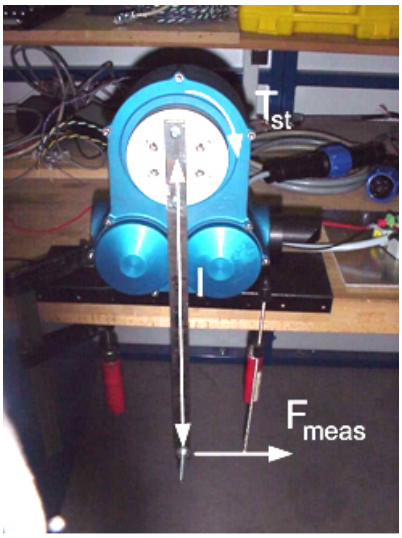

Figure 3.5: Calibration procedure

$$T_{st,meas} = F_{meas} \cdot l \text{ where } l\text{: Beam length} \tag{3.8}$$

The measured torques at different currents (I_q) are fitted by a linear fit to retrieve the torque constant.

The results of the steering robot are shown in figure 3.6. The figure shows that the delivered torque is far less than the ideal torque.

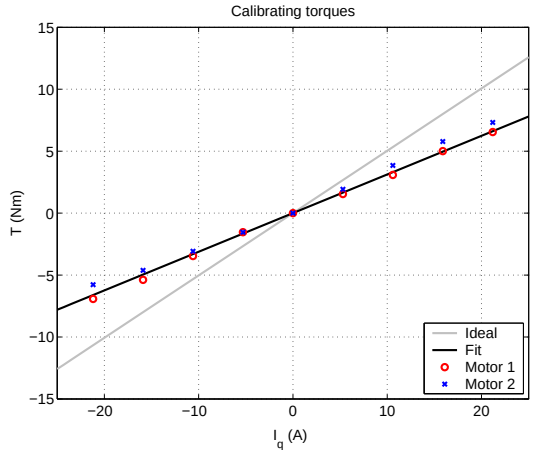

Figure 3.6: Calibration procedure results

3.2.2 Performance

Obviously the controller will operate correctly in steady state, since the produced torque in steady state is used to design the controller. However if the torque control is used at high speed the produced torque can decrease dramatically. This loss will not be corrected by the controller, since it is designed as a feedforward controller without feedback. Torque feedback will not be used to improve performance, since the produced torque can not be measured quick enough and it will not remove the cause of the problem.

The decrease in performance at high speed has two causes:

- Sample frequency of the controller
- Computation delay

The controller is a zero-order-hold controller. Each time sample one control action is computed and is used during that particular time sample. This is not a problem, when the actuator speed is significantly lower than the sample frequency. However if the actuator speed gets close to the sample frequency, the produced torque drops during a time sample. The control action is based on the actual actuator angle and if the angle changes too much during a time sample, the produced torque will be significantly smaller.

For example, if the sample frequency (f_s) is 25 times higher than the actuator speed (f_{act}) and the actuator has 6 pole pairs (n_p), the electrical angle rotation ($\Delta\gamma$) will be approximately 90 degrees according to (3.9).

$$\Delta\gamma = n_p \cdot \frac{f_{act}}{f_s} \cdot 360 \tag{3.9}$$

In other words, at the end of the time sample no torque is delivered by the actuator. In figure 3.7 (left) the produced torque is shown. The effective torque is indicated by the dotted line.

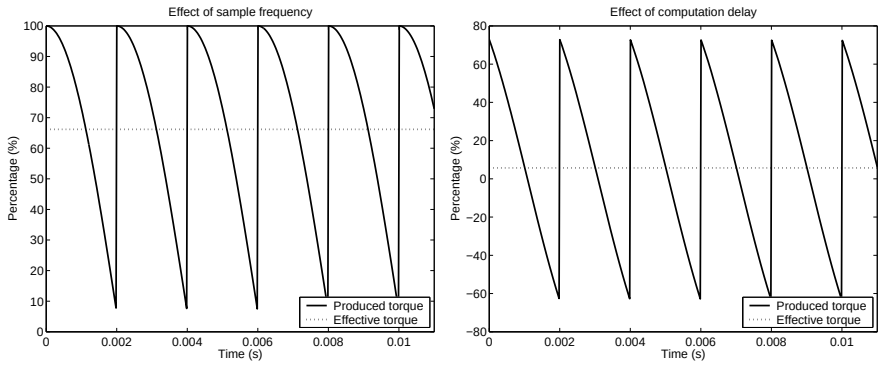

Figure 3.7: Effect of sample frequency (left) and computation delay (right) on produced torque

Similar problems occur if the computation delay is large compared to the actuator speed. The control action no longer corresponds to the current actuator angle and the produced torque will be less. To illustrate the effect of computation delay, a half time sample delay is introduced in the previous example. The produced torque is shown in figure 3.7 (right). The effective torque (dotted line) is almost zero.

Both effects can be combined to compute the effective torque percentage (T_{eff}). The average value is computed during one time sample.

$$T_{eff} = 100\% \cdot [\int_0^{t_s} cos(n_p\ f_{act}\ 2\ \pi(t + t_{delay}))\ dt]/t_s \tag{3.10}$$

In figure 3.8 the combined effect of sample frequency and computation delay on performance is shown. The actuator again has six pole pairs.

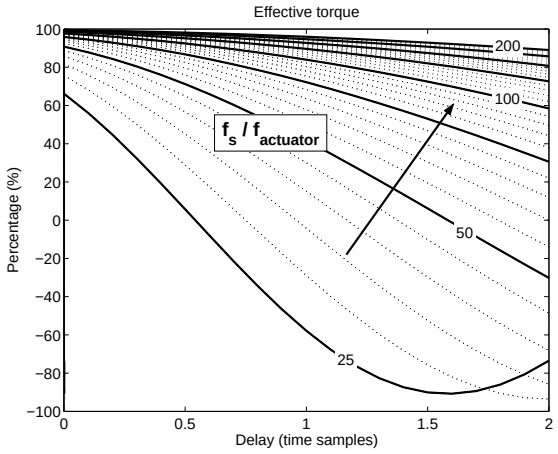

Figure 3.8: Combined effect of sample frequency and computation delay on produced torque

The steering robot's maximal rotation speed needs to be 1000 deg/s. To achieve this, the maximal frequency of the actuator needs to be 17 Hz. The current control system (MACS) runs at 500 Hz and this can not be increased further. The computation time can be assumed to be at least half the sample time, because the controller runs at full speed. According to figure 3.8, the performance is not acceptable at maximal rotation speed. The performance should be close to 100%, since everything else is pure loss.

The controller's sample frequency is insufficient to provide smooth control. In [DV04] a solution to this problem was proposed, but unfortunately it did not work in reality. The solution was to generate the sine commutation by using a synthesizer chip. This component can generate sinusoidal signals at a high frequency (25 MHz). The controller would only have to calculate amplitude, frequency and phase information at a much lower frequency and the synthesizer would use this information to generate a smooth sinusoidal signal. However to calculate the parameterized sine, the rotation speed is needed. This can not be

measured with reasonable accuracy, making it impossible for the synthesizer to compute a smooth sinusoidal signal.

The only way to increase the performance at high rotation speed, is to increase the ratio between controller's sample frequency and the actuator frequency. A new generation MACS is needed or another control system has to be used, which can be run at a higher sample frequency. It is also possible to redesign the system to reduce the actuator speed, e.g. a lower steering ratio.

3.3 Position controller

In the first section the design of the position controller is described. In the second section the performance is analyzed in simulation.

3.3.1 Design

The position controller consists of a torque controller and a Proportional-Integral-Derivative controller (PID controller). The PID controller computes a torque based on the error in the measured angle and the angle setpoint. This torque is then used by the torque controller developed in section 3.2. The Simulink model is shown in figure 3.9.

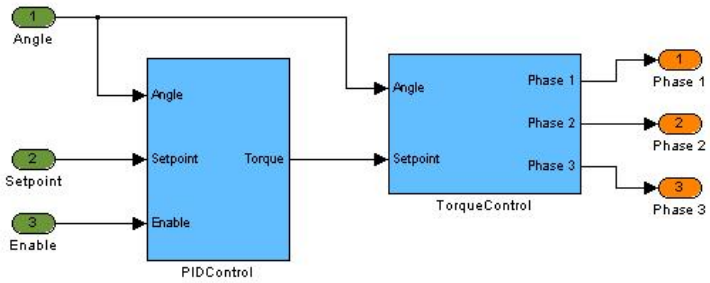

Figure 3.9: Position controller block in Simulink

One of the most basic position controllers is used. Nevertheless it is expected that the performance of the position controller will be sufficient, because the required steering frequencies are relatively low (up to 2 Hz).

To design the PID controller, Ziegler-Nichols tuning is used [FPE94]. Ziegler and Nichols wanted to specify satisfactory controller settings based on simple experiments without having to obtain the complete system dynamics.

One of the methods is based on evaluating the system at the limit of stability. The system is controlled by a proportional controller only. The gain of the controller is increased until the system becomes marginally stable. The resulting gain is called the ultimate gain (K_u). This tuning method also needs the period of oscillation, which is called the ultimate period (T_u).

With both parameters known a PID controller ($C(s)$) can be designed based on (3.11) and table 3.1. The proposed controllers must be seen as good average controllers, which are a good starting point, but need to be fine-tuned.

$$C(s) = K(1 + \frac{1}{T_i \cdot s} + T_d \cdot s) \tag{3.11}$$

Type controller	*Optimum gain*
Proportional	$K = 0.5 \cdot K_u$
PI	$K = 0.45 \cdot K_u$
	$T_i = \frac{5}{6} \cdot T_u$
PID	$K = 0.6 \cdot K_u$
	$T_i = 0.5 \cdot T_u$
	$T_d = \frac{1}{8} \cdot T_u$

Table 3.1: Ziegler Nichols controller design estimates

Because the produced torque is bounded, care must be taken with the integrator action. If the actuator saturates, the integrated value keeps on growing. This results in a big overshoot to get the integrated value close to zero again. The performance decreases and can even become instable. This phenomenon is called *integrator windup*.

To remove this phenomenon, the integration action is turned off, when the actuator saturates. This needs to be done, if the error and total actuator action have the same sign (i.e. the torque tries to minimize the error). The resulting PID controller is shown in figure 3.10.

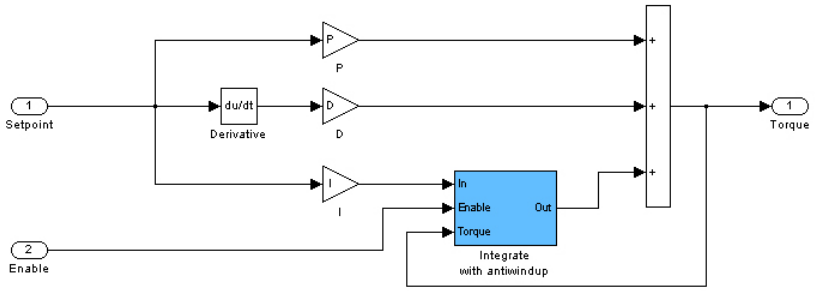

Figure 3.10: PID controller block in Simulink

3.3.2 Performance

The position controller of the steering robot must be a generic controller. It must be possible to use the steering robot in different vehicles under different

circumstances without adjusting the controller. When the steering robot is fitted, it must be operational at once with reasonable performance.

Due to the problems with the steering robot, it was not possible to design the position controller in reality and test the performance. However, to illustrate the proposed approach, the PID controller can still be designed in simulation.

To be able to design the PID controller, a simulation model needs to be built in Simulink with the dynamics of the controlled system. The simulation model consists of the control system, steering robot dynamics and a representative steering mechanism. These submodels are described in detail in chapter 6. The generated tyre forces are regarded as disturbances on the controlled system and will be neglected in the first controller design attempts. The setup is schematically shown in figure 3.11.

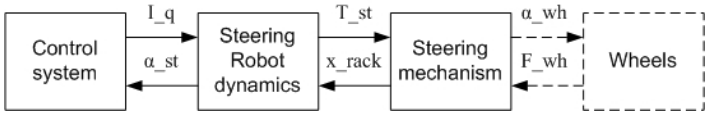

Figure 3.11: Simulation model

The PID controller can now be designed using Ziegler-Nichols tuning (see previous section). The steering angle target is set at 40 degrees, which is an average steering angle under normal driving circumstances. In figure 3.12 the response of a marginally stable proportional controller is shown. The gain (K_u) and period (T_u) of this controller are used to estimate the parameters of the PID controller based on table 3.1. The response of the resulting PID controller is also shown in figure 3.12.

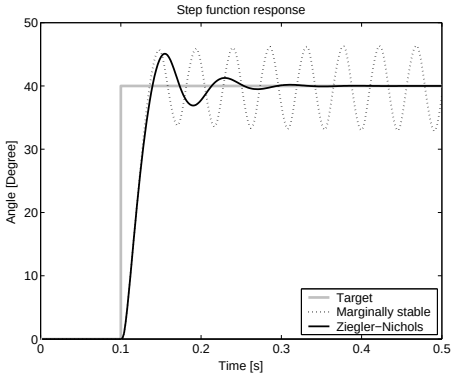

Figure 3.12: Ziegler-Nichols tuning (Step function response)

Although the response of the PID controller is satisfactory at first glance, it is no longer acceptable if the tyre forces are taken into account. The relatively

high tyre forces result in an angle offset[2], which is shown in figure 3.13.

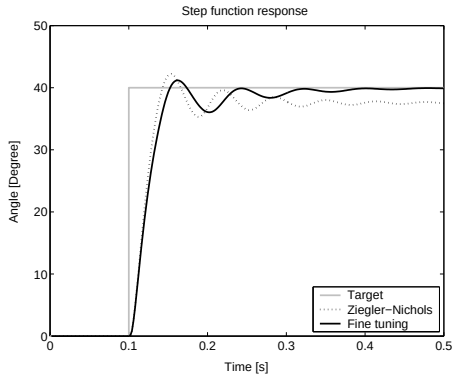

Figure 3.13: Fine-tuning (Step function response)

The angle offset can be removed. The parameter to remove this offset is the integral control action (T_i). This control action compensates for the accumulated error between actual angle and angle setpoint. If the integral control action (T_i) is increased, the target will be reached earlier. Furthermore the other parameters are also fine-tuned to dampen the controller's response. The fine-tuned PID controller is also shown in figure 3.13.

Figure 3.14 shows the response to a continuous sinusoidal steer input. In this figure angle, angular velocity and the applied steering torque are shown. Although the frequency of around 3 Hz is extremely high for steering input, the position control is still accurate.

[2]In this simulation the forces acting on the steering rack are simplified by using two springs on each side of the steering rack to reduce simulation times. These springs generate forces comparable to the forces generated by complex vehicle models, however it is only valid for small steering angles.

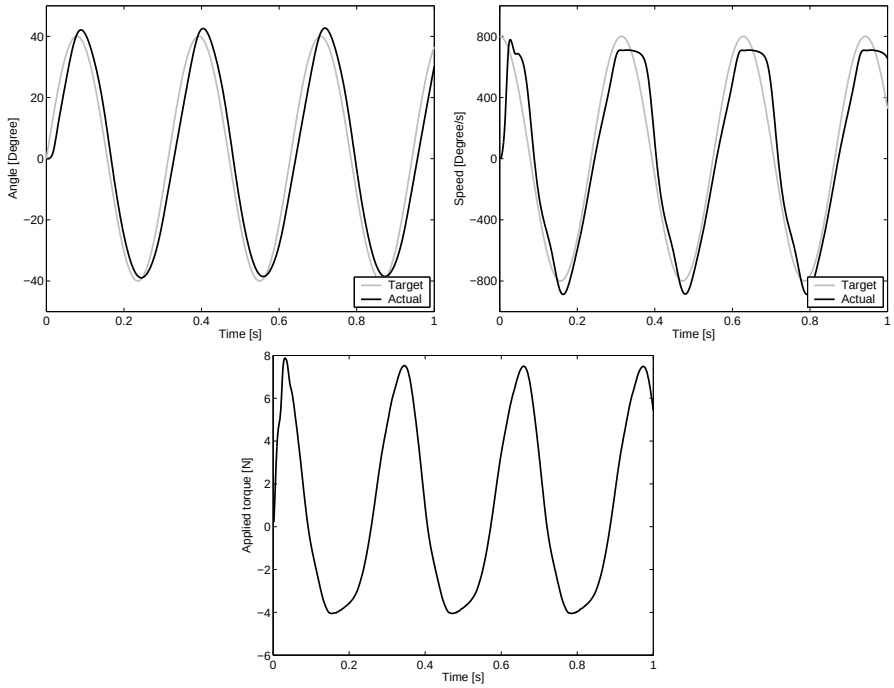

Figure 3.14: Continuous sine response

Chapter 4

Safety-critical hardware design

'Software does not fail, hardware always fails'

Paul Niquette

In this chapter different strategies are explained on how to make the hardware design of chapter 3 more safe. The focus for safety-critical hardware design will be on fault tolerance. First a general introduction is given to safety. In the next section the different fault-tolerant hardware strategies are explained and in the last section the fault-tolerant strategies are applied to the GDA.

4.1 Introduction

A system is said to be *safe*, if it will not endanger human life or the environment. The ability of a system to fulfil its safety requirements is limited by the presence of faults.

A *fault* is defined to be any kind of defect in the system. Faults can be either systematic (design faults) or random. Random faults are associated with hardware component failures. Although hardware component failures can also be caused by design faults, it is mostly caused by random faults, such as a break in the wiring, loose connectors, shorts, etc.

A fault can lead to a *system failure*, i.e. the system requirements can no longer be fulfilled. In a safety-critical system, such as the GDA, the risk of safety-critical failures needs to be reduced to a minimum, such that life-threatening situations can be regarded as eliminated.

The risk of failure can be reduced in different phases in system lifetime using different reliability strategies (see figure 4.1). *Fault prevention* techniques can

be used before the system becomes operational and tries to reduce the number of systematic faults being present. Fault prevention can be split in two parts *fault avoidance* (design phase) and *fault removal* (test phase). *Fault tolerance* techniques try to reduce the effect of faults when the system is operational for both the random and the remaining systematic faults.

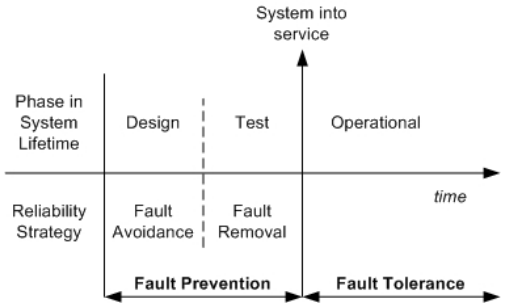

Figure 4.1: Reliability strategies during system lifetime [AL90]

Fault avoidance tries to reduce the number of systematic faults during design time. One way to reduce faults is by choosing a suitable design methodology to tackle the complexity of software and/or hardware design. An example of such a design methodology is the methodology discussed in chapter 5: *formal methods*. Another way to reduce faults is by selecting the proper techniques and technologies. By selecting reliable components the number of faults will certainly be reduced.

Fault removal tries to reduce the number of systematic faults by testing. Usually faults remain in the system, especially in complex systems. Test procedures can be used to find faulty hardware components or to detect remaining design faults. In chapter 6 *fault seeding* will be used to test safety. Different component failures will be presented to the system to see if the safety requirements can still be fulfilled, i.e. if the system remains safe.

Fault prevention is used to remove systematic faults. However it is impossible to remove random faults, since hardware components can never be 100% reliable. To increase reliability of hardware components fault tolerant strategies are used. Different strategies are explained in section 4.2.

The reliability of hardware components during its lifetime changes. The failure rate characteristic, also known as the 'bathtub' curve [STO99], is shown in figure 4.2.

In the first period failure rate is high due to undetected manufacturing defects. At the end of the components life it decays, resulting in a rising failure rate. In between the component has a more or less constant failure rate (λ). This failure rate will be used to characterize the reliability of a hardware component.

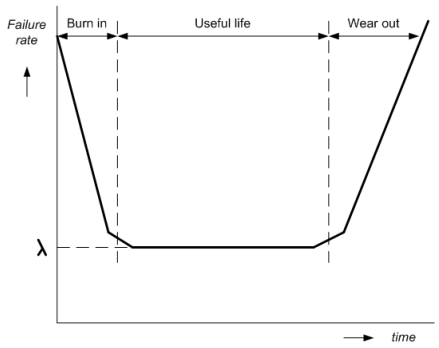

Figure 4.2: Component failure rate against time

4.2 Fault-tolerant strategies

Fault-tolerant strategies are used to cope with faults, when the system is operational. Fault-tolerant strategies to improve hardware safety are based on redundancy. Hardware components are duplicated, so that duplicated components can take over tasks, when others are failing. Although the number of faults increases due to the hardware redundancy, the number of failures decreases in general. Hardware redundancy increases reliability (less failures in general) and increases safety (less safety-critical failures). The strategies can be divided in two major groups: *static redundancy* and *dynamic redundancy*.

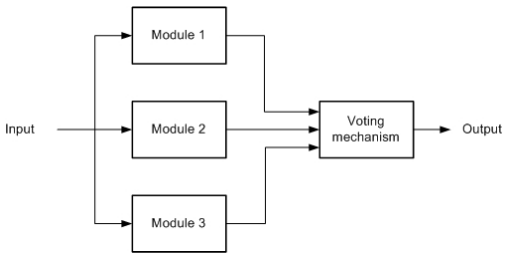

Figure 4.3: Static redundancy (TMR)

Static redundancy uses a voting mechanism and a number of redundant hardware components, which are performing the same tasks in parallel. In figure 4.3 the most basic static redundant setup is shown: Triple Modular Redundancy (TMR). In this setup three modules are performing the same task and their output is compared by a voting mechanism. The majority view is then taken by the voter, which *masks* a possible fault on a single module.

In the previous setup the voting mechanism can still be a potential risk, since it can also fail. To reduce the risk of a failing voting mechanism, it needs to be duplicated as well. An improved TMR setup is shown in figure 4.4. Note that this setup only works if the next component is able to receive three inputs or has a similar setup where the three outputs are used as inputs.

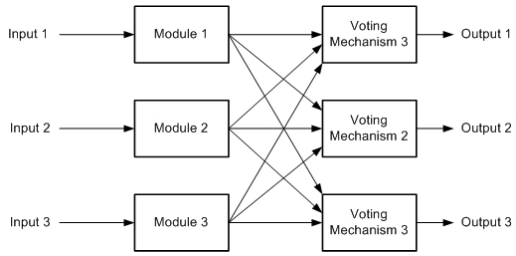

Figure 4.4: Static redundancy (TMR with multiple voters)

Dynamic redundancy also uses a number of redundant hardware components, but it tries to detect faults and reconfigures the setup. Unlike static redundancy it uses only one component at the same time. Another component is used as a spare and its output will not be used until a failure is detected on the main component. This spare can either be a cold or a hot stand-by, based on the desired power consumption and reconfiguration time. The most basic setup of dynamic redundancy is shown in figure 4.5.

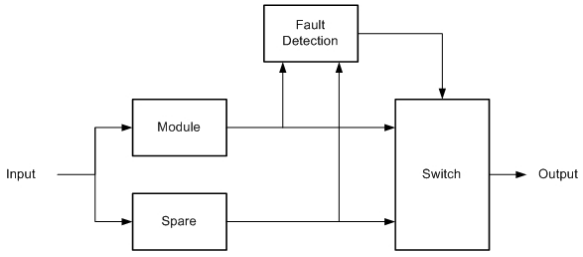

Figure 4.5: Dynamic redundancy (single spare)

The intention of dynamic redundancy is to use less hardware components compared to static redundancy to achieve the same level of fault tolerance. This makes the setup cheaper and it reduces power consumption, since it has fewer components. However dynamic redundancy lacks the ability to mask faults. It costs time to detect faults and reconfigure the system. Another problem is the fact that it is not always easy to detect component failure. The success of dynamic redundancy depends heavily on the ability to detect faults and failure rate of the fault detection mechanism.

The redundancy strategies can be adjusted to cope with more than one component failure by using additional redundant components. However it is also possible to use *hybrid redundancy*, which combines the advantages of both redundancy strategies. It can mask faults without a large amount of redundant hardware components and the fault detection is a lot easier, since the voter can be used for that purpose. A setup is depicted in figure 4.6 to cope with two component failures[1].

[1]This can only be guaranteed under the assumption that the second component failure occurs after reconfiguration has succeeded.

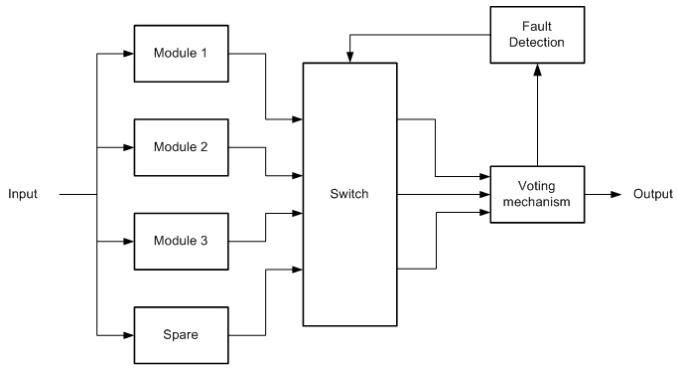

Figure 4.6: Hybrid redundancy (TMR with single spare)

Fault tolerance in general consists of three phases:

- *Fault detection*
  The first phase is to detect the faults. If the fault is not detected, it can not be handled in the first place. Different checks can be done to detect faults. Various examples are given below:

  - *Signal comparison*: Functionality is duplicated to be able to compare outputs and conclude if one is faulty.
  - *Watchdog timing*: It checks if a component is still operational.
  - *Plausibility checks*: It checks if a component has produced a reasonable output, for example if the output is within a certain range.
  - *Loopback testing*: The communication is checked by returning the received data to the sender. The returned data can then be compared with the sent data by the sender.
  - *Input checks*: It checks if the input is acceptable to produce output.

- *Fault recovery*:
  If the fault is detected, the fault can be handled properly to prevent system failure. If in a dynamic redundant system one component fails, this fault can be prevented from becoming a system failure by using the spare component instead. In a static redundant system the failing component will be masked by taking the majority view.

- *Fault treatment*
  The last phase is fault treatment. Fault recovery makes sure that the fault is properly handled, but this does not mean that the defect is removed. The defect need to be removed to guarantee a safe system. For example if one hardware component breaks down in a TMR design, this fault is handled. The next failing component however can not be handled. The safety of the system can not be guaranteed as long as the failing component is not replaced by a new one.

4.3 Fault-tolerant GDA

The GDA needs to be fault-tolerant. The safety must be guaranteed for single-point failures. If a single-point failure is detected in the system, the GDA is supposed to continue in a safe error mode, which will be designed in chapter 6. The GDA will be prevented from further utilization, until the component is either repaired or replaced. The possibility for a second component failure to occur is therefore limited and will be ignored.

To achieve fault tolerance the safety-critical hardware components need to be duplicated. The safety-critical hardware components are those components, which are needed for control: angle sensor, actuator, servo amplifier and control system.

4.3.1 Angle sensor

Triple Modular Redundancy is used for the angle sensor. The used angle sensors are relatively cheap and small, which does not limit the number of redundant components to be used. It is also easy to implement the voting mechanism, since voting can be done in software on the control system by taking the median of the three angles. The median is easy to calculate and it rules out the effect of the incorrect value. The fault-tolerant sensor design is shown in figure 4.7.

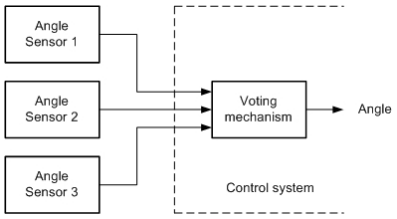

Figure 4.7: Fault-tolerant sensor design

4.3.2 Actuator and servo amplifier

For the actuator only dynamic redundancy can be used, but it is adapted to use the potential of both actuators. Unlike using a single spare component waiting to take over, both actuators are used simultaneously under normal conditions. If an actuator failure occurs, the other actuator tries to maintain safety. Due to the packaging and performance requirements, it is not possible to have spare actuators in the system.

Actuator failure can be detected by the servo amplifier. In general servo amplifiers have onboard functionality to protect against over/undervoltage problems, short circuits, overheating, etc. Actuator failure will be detected by the servo amplifier as a short circuit between motor power outputs and therefore actuator failure will be part of the fault detection of the servo amplifier. Because a servo amplifier is also responsible for one particular actuator, a servo amplifier and

the corresponding actuator are treated as one module, which is duplicated. The fault-tolerant actuator design is shown in figure 4.8.

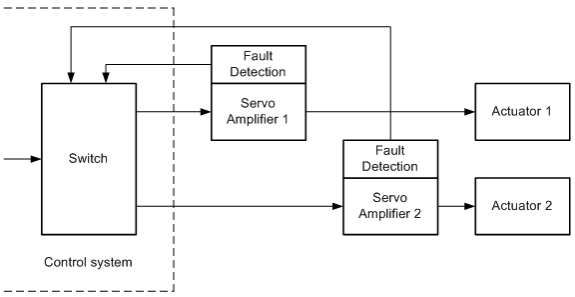

Figure 4.8: Fault-tolerant actuator design

The 'switch' is placed within the control system for two reasons. First of all no hardware is needed to implement the switch. Secondly and most importantly the switch is not a normal switch. During normal operation both actuators are performing half the control job. Furthermore both actuators need their own current setpoints. This makes the implementation in hardware more difficult.

4.3.3 Control system

The only component left is the control system. Static redundancy would be best to improve fault tolerance. This strategy will not introduce a control gap during failure, since no time is needed to switch between components. However it uses three control systems, which is a relatively expensive part of the GDA. Dynamic redundancy will be used for the control system, because the introduction of a small control gap during failure will not lead to dangerous situations.

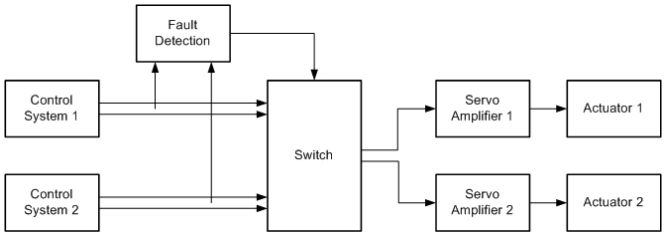

Figure 4.9: Fault-tolerant control system design (Single spare)

In figure 4.9 the conventional setup is shown using a single spare. In this setup a complex hardware component is needed, which is responsible for the switch and the fault detection (e.g. alive mechanism). This component also needs to be made fault tolerant to reduce the safety risks. If it fails, the safety can not be guaranteed, because no control can be done at all.

Another possibility is to give a control system the responsibility for one particular actuator. This setup is comparable to the actuator setup and is shown in figure 4.10. The fault detection is done by the other control system and the switch is distributed over the control systems.

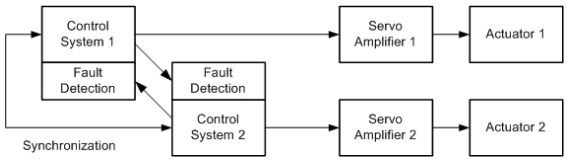

Figure 4.10: Fault-tolerant control system design (Dual control)

Both designs have advantages and disadvantages. The advantages of the single spare setup are given below:

- *Software complexity*: The software is less complex, since the tasks are not distributed over both control systems.
- *Safety after fault recovery*: This setup will handle the second failure better, since control system safety is independent of actuator safety. For example if one control system fails and is replaced by the other, than actuator safety is not effected.
- *Performance after fault recovery*: If the control system is replaced after it failed, both actuators can still be used.
- *Power consumption*: Power consumption is significantly reduced, if a cold standby is used. Unfortunately the control gap will be lengthened by a cold standby.

The dual control setup has the following advantages:

- *Performance in control gap*: If one control system is failing, the other control system is still doing 50% of the job. This means that during failure at least half of the required torque is delivered. However it is possible that the failing control system is thwarting.
- *Flexibility*: The design is more flexible, since it is implemented in software.
- *Hardware complexity*: The hardware design is more transparent, because one control system is responsible for one actuator and no (safety-critical) additional hardware is needed to implement the fault detection and switch.

The dual control setup will be used for the GDA, since its behaviour in the control gap is superior and no additional hardware is needed. Furthermore the safety and performance after fault recovery are not that important due to the introduction of the error mode.

Chapter 5

Safety-critical software design

'A computer program does what you tell it to do, not what you want it to do.'

Greer

This chapter describes the software design and verification of the control system using redundant hardware. Formal methods will be used to guarantee the requirements of the GDA's software. First an introduction is given on safety-critical software design and how formal methods fit in. In the next sections formal methods are used to specify, design and verify the GDA's software. In the final section the designed software is implemented in Simulink.

5.1 Introduction

Unlike hardware design, the only possible causes for software failure are design faults. This can either be a high-level (e.g. C# code and Matlab models) or a low-level (machine language) design fault. Duplication of software can not be applied, because identical software will always give identical results and these faults will not be detected. Therefore the fault-tolerant strategies described in section 4.2 will not work for software.

In literature the following fault-tolerant strategies are proposed to increase software safety:

- *Exception handling*: Exception handling can intercept software errors. These errors can then be avoided in code. However not all errors can be accounted for, since it is nearly impossible to handle all.
- *N-version programming* [AVI85]: Different code is used to perform the same tasks. This strategy is similar to the static redundancy strategy in section 4.2. N-version programming can be used to remove lower-level

design faults by using different processors and different compilers. Literature proclaims that this can also be done with high-level programming by using different design teams who implement multiple versions of the same algorithm. However this is disputed in [STO99], since research shows that different design teams are likely to make similar design faults.

- *Recovery blocks* [AL90]: If the first algorithm does not succeed (i.e. gives an acceptable result), the second (and different) algorithm tries to succeed and so on. This strategy is comparable to the dynamic redundancy strategy in section 4.2. Recovery blocks are not commonly used.

The difference between hardware and software fault tolerance is the type of redundancy used. Software fault-tolerant strategies use *heterogenous redundancy*, while hardware fault-tolerant strategies use *homogenous redundancy*. In general there is no reason to use less reliable hardware components, if component failure is random and independent. It only reduces the overall reliability of the system.

Although the fault-tolerant strategies mentioned above can improve software safety, the main focus need to be on fault prevention. Software failures are due to design faults and these need to be reduced during design time. The need for fault-tolerant strategies could be removed entirely for software design. At least in theory it could.

Fault avoidance can be done in many different ways (see section 4.1). During this thesis *formal methods* are used to design the software. Formal methods are mathematical techniques to specify, design and verify the system. These techniques will be explained in detail in the next sections and will be applied to the control system software. First a description of the system is made. The interface of the control system is described and how the signals should effect the control system. In the next sections the requirements are summarized and the specification is built in a process algebraic language. Finally the requirements are converted to modal logic and are verified.

The use of formal methods also has some disadvantages. The users have to have a high degree of mathematical ability, especially to verify the requirements and to keep the system manageable. Secondly the current tools can not cope with complex systems.

Fault removal is the last phase in the development of software. Although it is impossible to test and debug all functionality, due to the complexity of software, it is a crucial part in software design. Especially when formal methods can not be used, it is the only way to check the requirements. If the requirements *are* verified, it is still important to check if the code is implemented properly according to the specification. Software test methods will not be further explained. The GDA as a whole will be tested in chapter 6.

5.2 Interface description

In this section the interface description of the control system is given. The architecture of the GDA using redundant hardware (see section 4.3) is schematically

shown in figure 5.1. Although the GDA uses three robots, only one robot will be used throughout this chapter. This will not effect the functionality of the GDA as a whole, because it only effects the number of signals. Certain signals need to be replaced by tuples of signals.

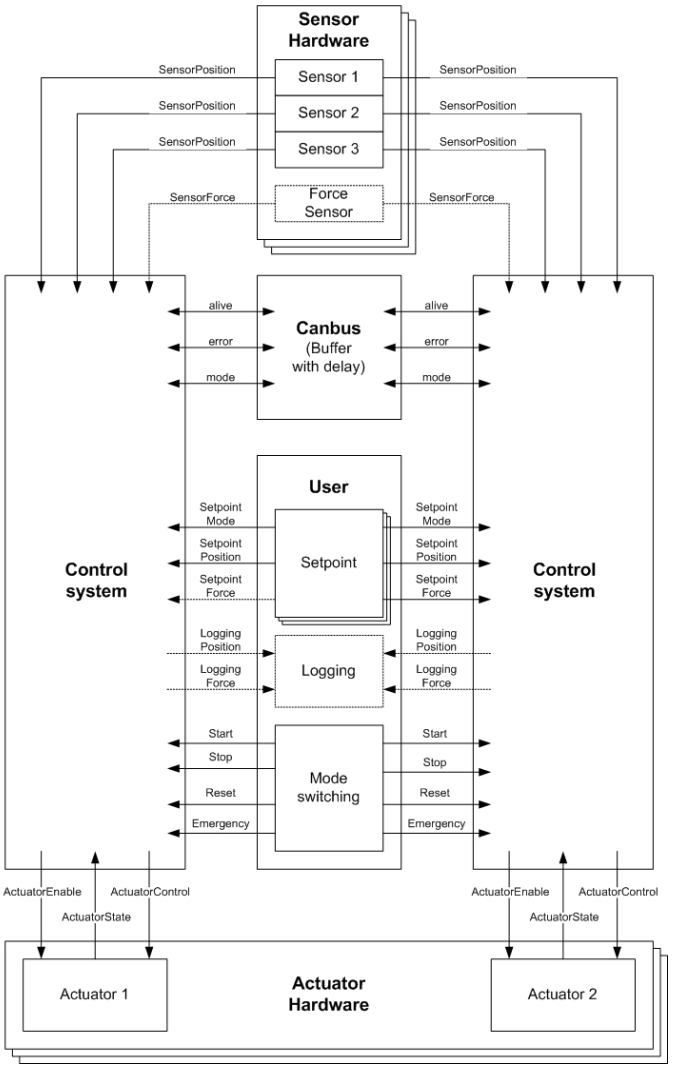

Figure 5.1: Architecture of the GDA using redundant hardware

5.2.1 Operational modes

First three conceptual modes are introduced to make the description clearer. The control system can be in one of three operational modes.

- *Menu mode*: The control system is in a mode, where none of the actuators is powered by the control system. The human driver is in full control of the vehicle.

- *Driving mode*: The control system drives the actuators and is in full control of the vehicle.
- *Error mode*: The control system signals an error. In this mode the actuators are utilized to control the vehicle safely to a stop in case of an error. The control system's behaviour in error mode will be further explained in chapter 6.

5.2.2 User interface

The user can give various signals to the control system. These signals are explained here.

The user can give signals effecting the mode of the control system:

- *Start signal*: If a start signal is provided in menu mode, the control system goes into driving mode. However if a sensor or actuator is failing operation, it stays in menu mode. Start signals in other modes are ignored by the control system.
- *Stop signal*: If a stop signal is provided in driving mode, the control system goes into menu mode directly. The driver will be in full control of the vehicle at once. Stop signals in other modes are ignored by the control system.
- *Reset signal*: If the control system detects a failing sensor or actuator in driving mode, it goes into error mode with an internal action (τ). If a reset signal is provided in error mode, the control system goes back into menu mode. Reset signals in other modes are ignored.
- *Emergency signal*: If an emergency signal is provided, the control system will go into error mode, whatever the mode it was in already. The control system's behaviour in such a situation is explained in chapter 6.

Summarizing this behaviour, results in the transition diagram shown in figure 5.2.

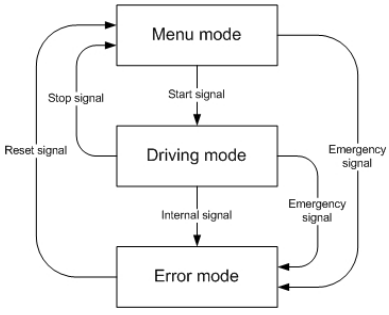

Figure 5.2: Modes of the GDA

The user can also provide and receive signals in driving mode:

- *Setpoints*: The user can give setpoints, which are used during driving mode to control the vehicle. There are different ways to provide setpoints (setpoint mode). However to reduce the complexity of the system, it is assumed that during driving mode only position setpoints are provided by the user.
- *Logging values*: In driving mode, real-time sensor signals are acquired by the control system either for control and logging (position) or for logging alone (force). These signals will be passed via the interface for logging. The logging signals are *LoggingPosition* and *LoggingForce*. These logging signals will not be modeled and analyzed.

5.2.3 Component interface

The interface between the control system and other components is explained in this section. These signals can directly be obtained from the hardware description of the GDA, introduced earlier.

- *Actuator signals*: The control system can send or receive the following signals to the actuator. The amplifiers will be part of the actuators for simplicity.
  - *ActuatorControl signal*: This signal contains the control values, which are sent to the actuator.
  - *ActuatorEnable* signal: The control system can enable the actuator, whenever it wants.
  - *ActuatorState* signal: The control system receives information, about the actuator's current state of operation.
- *Sensor signals*: The control system receives several sensor values:
  - *SensorPosition signals*: Three signals are provided, containing three independent position measurements.
  - *SensorForce signal*: Only one force measurement is provided. This signal will not be modeled and analyzed, because it is not used by the control system and will only be passed on for logging purposes.

5.2.4 Canbus interface

The communication between the two control systems will be via a canbus. The interface with the canbus is explained in this section. The communication signals are the following:

- *Mode signals*: The mode of the control systems are communicated over the canbus, to be able to synchronize modes.
- *Error mode signals*: The detected error of the control systems are sent over the canbus, to be able to adjust the error mode control based on the detected errors.
- *Alive signals*: Alive signals are communicated to detect if the other control system has crashed.

5.3 Requirements

The control system has the following requirements on its behaviour:

- *Deadlock freeness*: The control system can never be in a state, where no action can be performed anymore.
- *Requirement I*: When in driving mode with hardware operating correctly and no user actions received, the control system stays in driving mode.
- *Requirement II*: When a start signal is provided in menu mode with the hardware operating correctly and no other user actions received, the control system goes into driving mode as fast as reasonably possible.
- *Requirement III*: When a stop signal is provided in driving mode and no other user actions are received, the control system goes back into menu mode as fast as reasonably possible.
- *Requirement IV*: When the emergency signal is sent and no other user actions are received, the control system goes into error mode as fast as reasonably possible.
- *Requirement V*: When one sensor, actuator or the other control system fails in driving mode and no user actions are received, the control system goes into error mode as fast as reasonably possible.
- *Requirement VI*: The modes of both control systems have to be synchronized. This means that if one control system goes into another mode, both control systems go into the same mode as fast as reasonably possible, unless one control system is not responding. Note that it does not have to be the initial mode. If both control systems are in menu mode and one control system goes into driving mode, it is allowed to synchronize in error mode.
- *Requirement VII*: During driving mode, a difference between sensor value and setpoint will result in actuator action in the same time step. Unless it goes into error mode first.
- *Requirement VIII*: An actuator pair is synchronized during driving mode. Both actuators perform a similar action at the same time and will not perform opposing actions.

These requirements can only be fulfilled, under the assumption that there is at most one component failing. This is a valid assumption, since component failures are independent of each other in this setup and after a component failure the GDA is prevented from further usage. Therefore it is highly unlikely for a second component failure to occur.

5.4 Specification

In this section the specification of the control system is built in μCRL. But first a small introduction is given to μCRL.

5.4.1 Introduction to μCRL

Process algebra has been developed to express concurrent processes algebraically in an attempt to study their behaviour. Classical process algebras like CCS, CSP and ACP have proven to be a good way of specifying, analyzing and verifying the behaviour of distributed systems.

In this thesis the specification language μCRL is used, it is an extension of the process algebra ACP [BW90]. The main difference between μCRL and the classical process algebras is the formal treatment of data in μCRL. This makes μCRL more expressive than the classical process algebras.

A specification built in μCRL consists of two parts: data types and processes. The formal description of the syntax and semantics of μCRL can be found in [GP95].

Data types are declared by the keyword **sort**. Elements of a data type are declared by the keyword **func**. The keyword **map** is used to declare the syntax of a function and the semantics are defined with **rew**. No predefined data types are present in μCRL.

As an example the data type Bool and the equality function on Bool is implemented in μCRL. The data type Bool needs to be specified for every μCRL specification.

```
sort  Bool
func  T,F:→Bool
map   eq: Bool×Bool→Bool
var   b:Bool
rew   eq(T,T) = T
      eq(T,F) = F
      eq(F,T) = F
      eq(F,F) = T
```

Processes can be built from actions, operators and processes. The keyword **proc** is used to declare processes in μCRL.

User-defined actions are declared by the keyword **act**. Each user-defined action can have one or more data parameters. Only two predefined actions are introduced in μCRL: δ and τ. The δ is used for an inaction to indicate *deadlock*, i.e. the process stops performing actions. The τ indicates an *internal action*, i.e. an action that can be abstracted from.

The standard process algebraic operators are $(+)$, $(\cdot)$ and $(||)$. The operator $(+)$ is used to indicate *non-deterministic choice*. $p + q$ means that process p or process q will be executed. *Sequential composition* is indicated by the operator $(\cdot)$. $p \cdot q$ means that first process p and then process q is executed. The operator $(||)$ is used to indicate *parallel composition*. $p \,||\, q$ means that process p as well as process q can perform actions at the same time. To be able to work with data two new operators are introduced in μCRL: **sum** and $(\ \lhd\ \rhd\ )$. The first

operator is introduced to represent a (possibly infinite) choice between processes of a specific data type. For example $\mathbf{sum}(i : \mathbb{N}, p(i))$ means that one specific process p can be executed with a random natural number as a parameter. The second introduced operator indicates *conditional choice*. $p \lhd b \rhd q$ means that if boolean b is true process p will be executed and otherwise process q will be executed.

Synchronization between actions is achieved by the keyword **comm**. Actions may only synchronize if the data parameters are equal. To force synchronization between two actions the keyword **encap** is used. Otherwise the actions can also execute on their own.

Another important operator is **hide**. With this operator it is possible to hide the actions, which are not important for the analysis of the specification. These actions will than be transformed to τ actions. For example, if only the behaviour of two specific actions needs to be analyzed, all other actions can be hidden. One can abstract away from all unimportant actions for a particular analysis.

As an example a memory process is implemented in μCRL. The memory can receive both T or F and can send the boolean value, which is present in the memory.

```
act   recv_Bool: Bool
      send_Bool: Bool

proc  Memory(f: Bool) =
      sum(f_new: Bool, recv_Bool(f_new)·Memory(f_new)) +
      send_Bool(f)·Memory(f)
```

5.4.2 Data types

In this section several data types are introduced. Since redundant hardware is introduced for safety reasons, datatypes are defined to index the hardware components: MacsIndex and SensorIndex. An actuator index is not needed for the two actuators, because each control system has its own actuator.

```
sort  MacsIndex
func  macs1,macs2:→MacsIndex
map   other: MacsIndex→MacsIndex
rew   other(macs1) = macs2
      other(macs2) = macs1

sort  SensorIndex
func  sensor1,sensor2,sensor3:→SensorIndex
```

For the hardware also data types are needed to represent the data received from the sensors and the actions performed by the actuators: SensorValue and ActuatorAction. The data type SensorValue only has three elements to represent all possible positions: `sensorValueL5` (negative position), `sensorValue0` (neutral position) and `sensorValueR5` (positive position). The data type SensorValue

also has functions to compute the median of the three sensor values and to detect a failing sensor. A failing sensor means that a sensor differs from the median by more than one step. This can only be modeled with at least three data elements, otherwise no unambiguous decision can be made whether or not a sensor is failing. Each motion can then result in the conclusion 'failing sensor', when this motion falls between two sensor readings. The data type ActuatorAction also has three elements to represent all possible actions: `actuatorActionL` (negative force), `actuatorAction` (no force) and `actuatorActionR` (positive force). The requirements can be validated by only using the direction of actuator actions and three sensor values.

```
sort  SensorValue
func  SensorValueL5,SensorValue0,SensorValueR5:→SensorValue

map   match: SensorValue×SensorValue→Bool
map   min: SensorValue×SensorValue→SensorValue
map   max: SensorValue×SensorValue→SensorValue

map   median: SensorValue×SensorValue×SensorValue→SensorValue
var   sv1,sv2,sv3:SensorValue
rew   median(sv1,sv2,sv3) =
      max(max(min(sv1,sv2),min(sv1,sv3)),min(sv2,sv3))

map   fault: SensorValue×SensorValue×SensorValue×SensorValue
      →SensorValue
var   svMedian,sv1,sv2,sv3:SensorValue
rew   fault(svMedian,sv1,sv2,sv3) = ...

sort  ActuatorAction
func  actuatorActionL,actuatorAction0,actuatorActionR
      :→ActuatorAction
```

For the interface with the user, two simple data types are introduced: UserAction and SetpointValue. The data type SetpointValue has three similar elements as the data type SensorValue.

```
sort  UserAction
func  userStart,userStop,userEmergency,userReset:→UserAction

sort  SetpointValue
func  setpointValueL5,setpointValue0,setpointValueR5:→SetpointValue
```

The current status needs to be saved for the behaviour of the control system in the next time sample. For this reason two data types are defined: Mode and Error. The data type Error is a summation of all possible errors that can occur: `errorSensor`, `errorMACS` (other control system has detected an error), `errorAlive`, `errorController` (control algorithm produces an error) and `errorActuator`. In this summation, the data elements are ordered by their priority and this ordering can be compared by a function. The data element `errorNo` is also added to indicate error mode by using the emergency button.

```
sort  Mode
func  modeMenu,modeDrive,modeError:→Mode

sort  Error
func  errorNo,errorSensor,errorMACS,errorAlive,
      errorController,errorActuator:→Error

map   higherPriority: Error×Error→Bool
rew   higherPriority(errorNo,errorNo) = F
      higherPriority(errorSensor,errorNo) = T
      ...
```

5.4.3 Processes

In this section the specification of the complete control system using redundant hardware is explained. The specification treated here is simplified, the complete μCRL specification can be found in appendix D.1.

The control system and its environment are modeled by five major processes, which operate in parallel: `MACS(macs1)`, `MACS(macs2)`, `Canbus`, `Hardware` and `User`. The process `Canbus` is modeled as a memory with a single delay. The processes `Hardware` and `User` can be used to model restrictions on the input signals.

```
proc  System =
      MACS(macs1) || MACS(macs2) ||
      Canbus ||
      Hardware || User
```

The two `MACS` processes are symmetrical, both performing the same actions. The control system performs its tasks sequentially and time-driven, a computation is done each time sample. A time sample consists of five subsequent processes: `Monitor_In`, `Position`, `Supervisor`, `Controller` and `Monitor_Out`. The process `MACS` then waits until it is allowed to start a new time sample: `Time`. The timing mechanism is treated later in this section. Each control system also has five internal memories to store variables, these memories are run in parallel with the control system.

```
proc  MACS(i:MacsIndex) =
      Monitor_In(i) · Position(i) ·
      Supervisor(i) · Controller(i) ·
      Monitor_Out(i) ·
      Time(i) · MACS(i)
```

The `Monitor_In` process monitors the other control system. First it checks if it is still operating by receiving alive signals over the canbus. If the other control system is no longer alive it sends an error to the memory `Memory_Error`, which is used by the `Supervisor` process. It also checks the mode and the detected

error of the other control system. If it is in error mode, it also sends an error to the memory `Memory_Error`. The error kept in this memory will only be changed by an error with a higher priority. The detected error of the other control system is kept in the memory `Memory_ErrorOther` and will be used by the `Controller` process. The `Monitor_Out` process makes sure the other control system gets the same information back.

The `Position` process computes the best position. The median is computed from the three sensor values, received from the `Hardware` process. Furthermore it checks whether or not a sensor is failing. These two values are kept in a memory: `Memory_SensorValue` and `Memory_SensorError`. These values can then be used in the processes `Supervisor` and `Controller`.

```
proc Position(i:MacsIndex) =
     sum(sv1:SensorValue, recv_Position_Sensor(i,sensor1,sv1) ·
       sum(sv2:SensorValue, recv_Position_Sensor(i,sensor2,sv2) ·
         sum(sv3:SensorValue, recv_Position_Sensor(i,sensor3,sv3) ·
           send_Position_SensorValue(i,median(sv1,sv2,sv3)) ·
           send_Position_SensorError(i,fault(median(sv1,sv2,sv3),sv1,sv2,sv3))
         )
       )
     )
```

The `Supervisor` process takes care of the computation of the operational mode. This responsibility can be split up in two parts (`Supervisor_Systemcheck` and `Supervisor_Mode`). First a system check is done to find out if all components (sensors, actuator and other control system) are still working properly. If one component is failing, the error is stored in the memory `Memory_Error`. Then the operational mode is determined.

```
proc Supervisor(i:MacsIndex) =
     Supervisor_Systemcheck(i) ·
     Supervisor_Mode(i)
```

The `Supervisor_Mode` process determines the operational mode from the previous mode, the system check and the user action:

- *Menu mode*: The mode is changed to driving mode, if a start signal is received. If however not all components are functioning, the mode remains the same and a reset signal is needed to clear the errors stored in the memory `Memory_Error`. An emergency signal changes the mode to error mode and a stop signal is ignored in menu mode.
- *Driving mode*: A stop signal changes the mode to menu mode. If an emergency signal is received or if not all components are functioning, the mode changes to error mode. The other two user signals are ignored.
- *Error mode*: This mode can only be changed by a reset signal and the mode change to menu mode.

```
proc  Supervisor_Mode(i:MacsIndex) =
      sum(m: Mode, recv_Spv_Mode(i,m) ·
        (
          (Supervisor_Menu(i) ◁ eq(m,modeMenu) ▷ δ ) +
          (Supervisor_Drive(i) ◁ eq(m,modeDrive) ▷ δ ) +
          (Supervisor_Error(i) ◁ eq(m,modeError) ▷ δ )
        )
      )

proc  Supervisor_Menu(i: MacsIndex) =
      sum(ua: UserAction, recv_Spv_UI(i,ua) ·
        (
          (Supervisor_Start(i) ◁ eq(ua,userStart) ▷ δ ) +
          (τ ◁ eq(ua,userStop) ▷ δ ) +
          (send_Spv_Mode(i,modeError)·send_Spv_Error(i,errorNo)
          ◁ eq(ua,userEmergency) ▷ δ ) +
          (send_Spv_Reset(i) ◁ eq(ua,userReset) ▷ δ )
        )
      )

proc  Supervisor_Start(i: MacsIndex) =
      sum(e: Error, recv_Spv_Error(i,e) ·
        (send_Spv_Mode(i,modeDrive)
        ◁ eq(e,errorNo) ▷
        errorMessage(e))
      )

proc  Supervisor_Drive(i: MacsIndex) =
      sum(e: Error, recv_Spv_Error(i,e) ·
        (
          sum(ua: UserAction, recv_Spv_UI(i,ua) ·
            (
              (τ ◁ eq(ua,userStart) ▷ δ ) +
              (send_Spv_Mode(i,modeMenu) ◁ eq(ua,userStop) ▷ δ ) +
              (send_Spv_Mode(i,modeError) ◁ eq(ua,userEmergency) ▷ δ ) +
              (τ ◁ eq(ua,userReset) ▷ δ )
            )
          )
            ◁ eq(e,errorNo) ▷
          send_Spv_Mode(i,modeError) · Supervisor_Error(i)
        )
      )

proc  Supervisor_Error(i: MacsIndex) =
      sum(ua: UserAction, recv_Spv_UI(i,ua) ·
        (send_Spv_Reset(i) · send_Spv_Mode(i,modeMenu)
        ◁ eq(ua,userReset) ▷ τ )
      )
```

The `Controller` process takes care of the control action in a specific mode:

- *Menu mode*: The controller does not have to perform any control actions.
- *Driving mode*: The controller computes a control action based on the computed sensor value and the setpoint. The controller can also give an error, if the control action can not be computed.
- *Error mode*: The controller computes an error mode control action, if the corresponding actuator is functioning properly. If the error mode control action can not be computed, the controller also gives an error.

Three dummy actions (`menu`, `drive` and `error`) are added to this process to make verification easier in section 5.6.

```
act   menu: MacsIndex
      drive: MacsIndex
      error: MacsIndex

proc  Controller(i:MacsIndex) =
      sum(m: Mode, recv_Spv_Mode(i,m) ·
        (
          (menu(i) ◁ eq(m,modeMenu) ▷ δ ) +
          (drive(i) · Controller_Drive(i) ◁ eq(m,modeDrive) ▷ δ ) +
          (error(i) · Controller_Error(i) ◁ eq(m,modeError) ▷ δ )
        )
      )

proc  Controller_Drive(i: MacsIndex) =
      sum(sv: SensorValue, recv_Ctrl_SensorValue(i,sv) ·
        sum(s: SetpointValue, recv_Ctrl_Setpoint(i,s) ·
          (
            send_Ctrl_ActuatorAction(i,action(sv,s)) +
            send_Spv_Error(i,errorController)
          )
        )
      )

proc  Controller_Error(i: MacsIndex) =
      sum(e: Error, recv_Spv_Error(i,e) ·
        (send_Ctrl_ActuatorAction(i,actuatorAction0)
          ◁ eq(e,errorActuator) ▷
        (
          send_Ctrl_ActuatorAction(i,actuatorActionError)) +
          send_Spv_Error(i,errorController)
        )
      )
```

During normal operation both control system will operate almost in turn and it is highly unlikely that one control system will compute 5 time samples before the

other computes another time sample. To be able to achieve this a timing mechanism is built, which will guarantee that both control systems will operate almost synchronous. The process `Time` of one control system sends actions `send_time` and the process `Time` of the other control system sends actions `recv_time`. The processes are synchronized by enforcing communication between these actions.

```
act     comm_time send_time recv_time
comm    send_time | recv_time = comm_time

proc    Time(i:MacsIndex) =
        (
          (
            send_time · send_time +
            send_time · send_time · send_time
          )
          ◁ eq(i,macs1) ▷
          (
            recv_time · recv_time +
            recv_time · recv_time · recv_time
          )
        )
```

The actions of a control system during one time sample are independent of the other except for the communication over the canbus. To reduce parallelism these actions are split up in two indivisible blocks using *binary semaphores* (see [DYK65]). This makes the verification in the next section a lot easier, since the time samples are now (almost) sequential. This new technique and its validity is further explained in appendix B. The process `MACS` is adjusted as follows:

```
proc  MACS(i:MacsIndex) =
      send_Semaphore(P) ·
      Monitor_In(i) · Position(i) ·
      Supervisor(i) · Controller(i) ·
      send_Semaphore(V) · send_Semaphore(P) ·
      Monitor_Out(i) ·
      send_Semaphore(V) ·
      Time(i) · MACS(i)
```

All processes and their communication are shown in figure 5.3. In this figure only one control system is shown, since their behaviour is completely symmetrical.

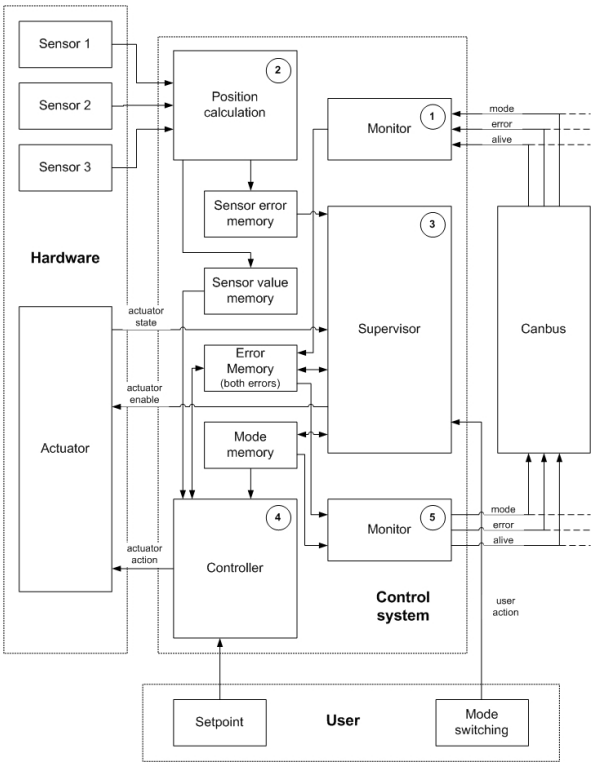

Figure 5.3: Overview of all processes in μCRL design

5.5 Analysis results

During verification of the requirements, some problems were found in the requirements and the original specification. These problems resulted in modifications to the requirements and the specification and are discussed in this section. In section 5.6 the final version of the specification is verified.

5.5.1 Problem 1

The first problem is related to the requirements. The last two requirements can *not* be fulfilled at the same time, since they are in contradiction. Every change in setpoint can lead to a situation with two actuators performing opposing actions, if the actuator is supposed to respond to a difference between sensor value and setpoint in the *same* time step. This contradiction is shown in figure 5.4. Both actuators try to turn left by performing force in that direction. If the setpoint is then changed to a neutral position, the actuators can momentarily perform opposing forces.

The specification built only fulfills requirement VII, but it is also possible to adjust the specification to fulfill the actuator synchronization requirement instead. Two options are given:

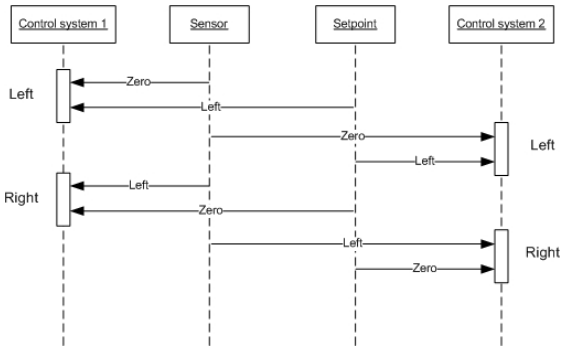

Figure 5.4: Example showing conflicting requirements

- *Communicate actuator actions over canbus*: This method results in a delay for *all* actuator actions. This is unacceptable according to section 3.2, since it will result in poor controller performance.

- *Don't allow opposing actuator actions on a single control system in subsequent time samples*: If the direction of the actuator action changes no actuator action is performed to prevent opposing forces. It only works if there are two time samples with no actuator action, this is shown in figure 5.5. This strategy will certainly have a negative effect on controller performance, especially with position control where many direction switches occur.

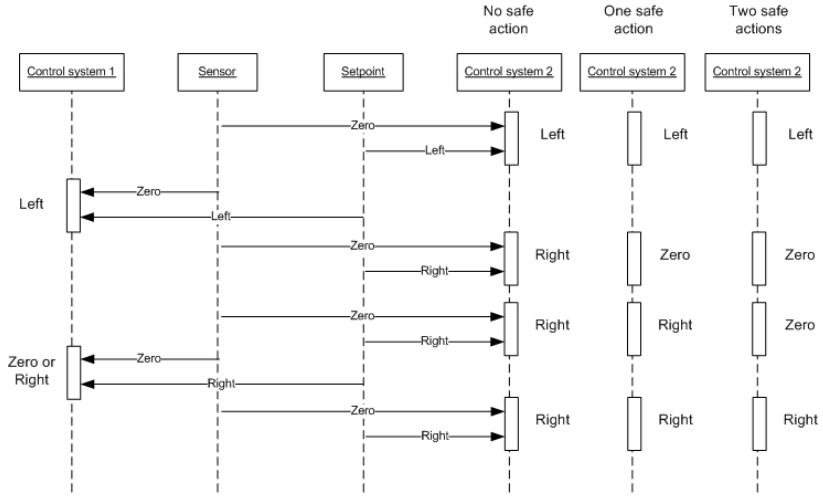

Figure 5.5: Second option to fulfill requirement VIII

The actuator synchronization requirement is dropped, because the negative effect on actuator synchronization is regarded as less important than the negative effect on performance.

5.5.2 Problem 2

The second problem is shown in UML in figure 5.6 (left). If a start signal is provided by the user in menu mode, the control system can decide to go into driving mode or stay in menu mode if it has detected an error. If one control system stays in menu mode while the other goes into driving mode, the modes could not be synchronized.

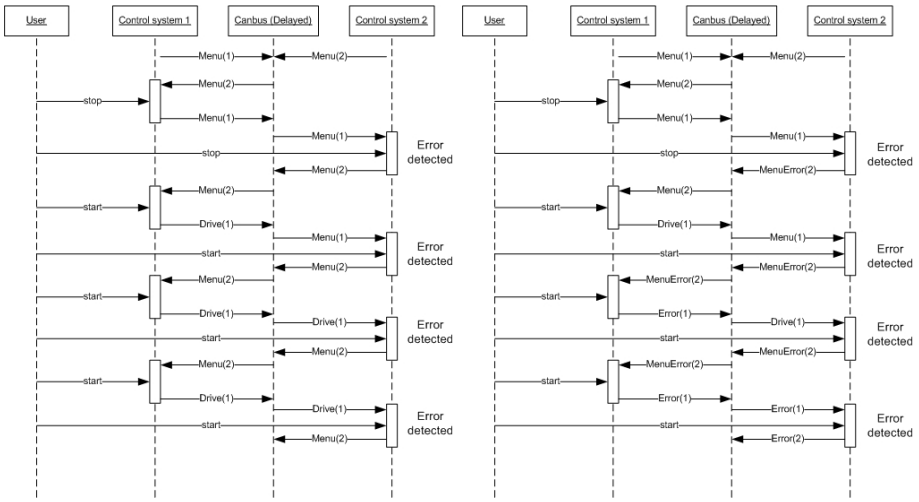

Figure 5.6: Problem starting driving mode: Original problem (left) and solution (right)

The problem is solved by synchronizing both control systems in error mode. Two modifications are needed to achieve this. First a special mode is needed to indicate a menu mode with detected errors, which is sent over the canbus. If a control system is in driving mode and receives a mode signal indicating menu mode with detected errors, the control system goes into error mode immediately. Secondly the specification is changed to make it possible to go to error mode from menu mode if a error mode signal is received from the other control system. A new datatype `ModeOther` needs to be introduced and the process `Monitor_In` needs to be adjusted.

```
sort  ModeOther
func  otherMenu,otherMenuError,otherDrive,otherError:→ModeOther
```

```
proc    Monitor_In(i:MacsIndex) =
        Monitor_In_Alive(i) ·
        sum(m: ModeOther, recv_Mon_Mode_Other(other(i),m) ·
          (
            ( τ ◁ eq(m,otherMenu) ▷ δ ) +
            (send_Spv_Error(i,errorMACS) ◁ eq(m,otherMenuError) ▷ δ ) +
            ( τ ◁ eq(m,otherDrive) ▷ δ ) +
            (send_Spv_Error(i,errorMACS) · send_Spv_Mode(i,modeError)
```

```
            ◁ eq(m,otherError) ▷ δ )
        )
    )
```

After these modifications the modes synchronize properly in error mode. This is shown in figure 5.6 (right).

5.5.3 Problem 3

The third problem is shown in UML in figure 5.7 (left). If a reset signal is provided in driving mode and one of the control systems detects an error, it is possible to have one control system in driving mode and the other in menu mode. The problem is caused by the fact that mode signals on the canbus can be overwritten. The error mode signal is overwritten by a menu mode signal and the other control system therefore stays in driving mode without ever noticing the error mode signal.

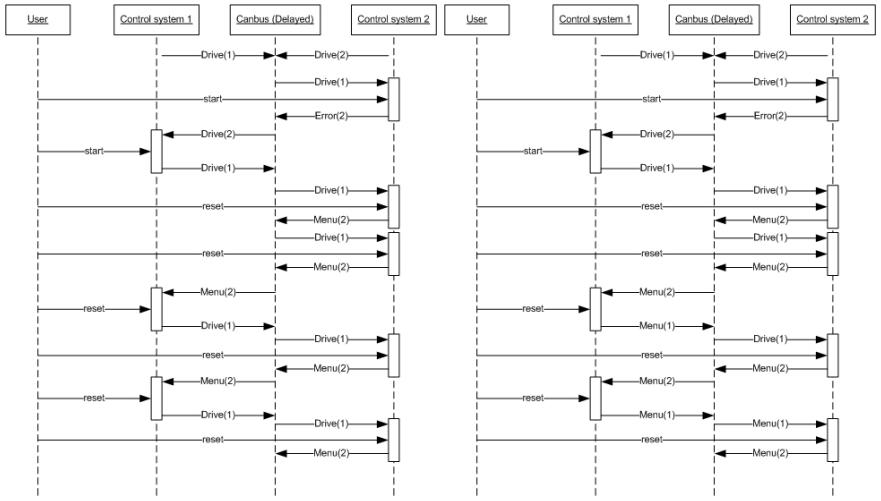

Figure 5.7: Problem with reset action: Original problem (left) and solution (right)

This synchronization problem is solved by changing the process `Supervisor_Drive` in such a way that the reset signals also results in a transition from driving mode to menu mode. The effect is depicted in figure 5.7 (right).

```
proc Supervisor_Drive(i: MacsIndex) =
     sum(e: Error, recv_Spv_Error(i,e) ·
       (
         sum(ua: UserAction, recv_Spv_UI(i,ua) ·
           (
             (τ ◁ eq(ua,userStart) ▷ δ ) +
```

```
            (send_Spv_Mode(i,modeMenu) ◁ eq(ua,userStop) ▷ δ ) +
            (send_Spv_Mode(i,modeError) ◁ eq(ua,userEmergency) ▷ δ ) +
            (send_Spv_Mode(i,modeMenu)τ ◁ eq(ua,userReset) ▷ δ )
          )
        )
          ◁ eq(e,errorNo) ▷
        send_Spv_Mode(i,modeError) · Supervisor_Error(i)
      )
    )
```

5.6 Verification

In this section the requirements will be verified. First some theory is presented about model-checking by using modal logics. The requirements can then be rewritten in modal logics and the transition system is generated. In the last section results of the verification are presented.

5.6.1 Introduction to model-checking

With the specification designed in the previous section, it is possible to generate a transition system and to verify the requirements on the specification. The transition system can be visualized by using one of the tools described in appendix C.3.

However in most cases the verification can not be done by hand, because the transition system is too complex. Especially when many parallel processes are introduced, the transition system grows exponentially. This problem is called the *state-explosion problem*. This phenomenon introduces a need for automatic verification of systems: model-checking. A model-checker is a tool which decides whether a given specification satisfies a certain requirement which is expressed by a logical formula ϕ.

All actions, which are not used in the requirements, are hidden. After abstracting away from the internal behaviour, a reduced *branching bisimilar*[1] transition system can be computed. The resulting transition system is a lot smaller than the original transition system, which increases the model-checker's performance.

There are several different modal logics introduced to express the requirements on the system. The most important modal logics are PLTL, HML, CTL and modal μ-calculus. A short introduction to these different logics can be fount in [MSS99]. Only modal μ-calculus will be treated here, since it is used by the main model-checkers and it is claimed to be the most powerful modal logic.

One specific form of modal μ-calculus is used, which is developed for the model-checker *Evaluator*[2]: regular alternation-free μ-calculus. The syntax of regular

[1] *Branching bisimilarity* is an equivalence on transition systems which preserves the validity of modal logic. A formal definition can be found in [FGR05]

[2] Model-checker in the Cæsar Aldébaran Development Package (see appendix C.2)

alternation-free μ-calculus is as follows [MS00]:

$$\alpha ::= a \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 \wedge \alpha_2$$
$$\beta ::= \alpha \mid \beta_1 \cdot \beta_2 \mid \beta_1|\beta_2 \mid \beta^*$$
$$\phi ::= \mathrm{F} \mid \mathrm{T} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid Y \mid \mu Y.\phi \mid \nu Y.\phi \mid \langle\beta\rangle\phi \mid [\beta]\phi$$

The action formulae α are built from action names a and the boolean operators ($\neg$), ($\vee$) and ($\wedge$). Furthermore the sign $\top$ is introduced to indicate the set of all actions. The regular formulae β are built from action formulae and the standard regular expressions operators ($\cdot$), ($|$) and (*). The operators denote concatenation, choice and transitive-reflexive closure respectively. For example $(a_1|a_2)^*$ means zero or more times a choice is made between action a_1 and a_2. The operator ($^+$) is also introduced. $(a)^+$ means at least one action a is performed.

The state formulae ϕ are built from propositional variables Y by using the standard boolean operators. Two operators are introduced for possibility and necessity. $\langle\beta\rangle\phi$ means 'it is possible to do a sequence of actions β to a state where ϕ holds' and $[\beta]\phi$ means 'ϕ holds in all states reachable by a sequence of actions β'. Least and greatest fixpoint operators are denoted by $\mu Y.\phi$ and $\nu Y.\phi$. With the fixpoint operators recursion could be introduced in the original versions of modal μ-calculus. However working with fixpoint operators is very tricky and hard to understand. Therefore the (*) is introduced in regular alternation-free μ-calculus, which is more intuitive. The fixpoint operators and (*) are related as follows:

$$\mu Y.(\phi \vee \langle\alpha\rangle Y) \equiv \langle\alpha^*\rangle\phi$$
$$\nu Y.(\phi \wedge [\alpha] Y) \equiv [\alpha^*]\phi$$

An enlightening introduction to modal μ-calculus and the fixpoint operators can be found in [BS01].

5.6.2 Expressing requirements in modal logic

Before expressing the requirements in modal logic, some macros are introduced to make the requirements more readable. First some action formulae are introduced. The action formulae can be used with or without control system index i.

- The action formula α_{user} is to indicate all user actions.

  $\alpha_{user}(i) \equiv \texttt{start}(i) \vee \texttt{stop}(i) \vee \texttt{emergency}(i) \vee \texttt{reset}(i)$

  $\alpha_{user} \equiv \alpha_{user}(\texttt{macs1}) \vee \alpha_{user}(\texttt{macs2})$

- The action formula α_{error} is to indicate the actions when a system error is detected. Since these errors are all kept in the memory `Memory_Error`, the internal communications can be used to indicate detected system errors.

  $\alpha_{error}(i) \equiv$
  $\texttt{comm\_Error\_Set}(i, \texttt{errorSensor}) \vee$
  $\texttt{comm\_Error\_Set}(i, \texttt{errorMACS}) \vee$

$\texttt{comm\_Error\_Set}(i, \texttt{errorAlive}) \vee$
$\texttt{comm\_Error\_Set}(i, \texttt{errorController}) \vee$
$\texttt{comm\_Error\_Set}(i, \texttt{errorActuator})$

$\alpha_{error} \equiv \alpha_{error}(\texttt{macs1}) \vee \alpha_{error}(\texttt{macs2})$

- The action formula $\alpha_{errorknown}$ is to indicate the actions when a system error is known. Since these errors are already in the memory `Memory_Error`, the internal communications can be used to indicate known system errors.

  $\alpha_{errorknown}(i) \equiv$
  $\texttt{comm\_Error\_Get}(i, \texttt{errorSensor}) \vee$
  $\texttt{comm\_Error\_Get}(i, \texttt{errorMACS}) \vee$
  $\texttt{comm\_Error\_Get}(i, \texttt{errorAlive}) \vee$
  $\texttt{comm\_Error\_Get}(i, \texttt{errorController}) \vee$
  $\texttt{comm\_Error\_Get}(i, \texttt{errorActuator})$

  $\alpha_{errorknown} \equiv \alpha_{errorknown}(\texttt{macs1}) \vee \alpha_{errorknown}(\texttt{macs2})$

- The action formula α_{mode} is to indicate all mode actions.

  $\alpha_{mode}(i) \equiv \texttt{menu}(i) \vee \texttt{drive}(i) \vee \texttt{error}(i)$

  $\alpha_{mode} \equiv \alpha_{mode}(\texttt{macs1}) \vee \alpha_{mode}(\texttt{macs2})$

Some system properties are expressed in regular formulae.

- The regular formula β_{menu} is to indicate each state reachable where both control systems are in menu mode. It states that after a random sequence of actions, the last two mode actions performed on both control systems are the `menu` actions.

  $\beta_{menu} \equiv \top^* \cdot$
  $(\texttt{menu}(\texttt{macs1}) \cdot (\top \wedge \neg\alpha_{mode}(\texttt{macs1}) \wedge \neg\alpha_{error}(\texttt{macs1}))^* \cdot \texttt{menu}(\texttt{macs2}) +$
  $\texttt{menu}(\texttt{macs2}) \cdot (\top \wedge \neg\alpha_{mode}(\texttt{macs2}) \wedge \neg\alpha_{error}(\texttt{macs2}))^* \cdot \texttt{menu}(\texttt{macs1}))$

- The regular formula β_{drive} is to indicate each state reachable where both control systems are in driving mode.

  $\beta_{drive} \equiv \top^* \cdot$
  $(\texttt{drive}(\texttt{macs1}) \cdot (\top \wedge \neg\alpha_{mode}(\texttt{macs1}) \wedge \neg\alpha_{error}(\texttt{macs1}))^* \cdot \texttt{drive}(\texttt{macs2}) +$
  $\texttt{drive}(\texttt{macs2}) \cdot (\top \wedge \neg\alpha_{mode}(\texttt{macs2}) \wedge \neg\alpha_{error}(\texttt{macs2}))^* \cdot \texttt{drive}(\texttt{macs1}))$

- The regular formula β_{error} is to indicate each state reachable where both control systems are in error mode.

  $\beta_{error} \equiv \top^* \cdot$
  $(\texttt{error}(\texttt{macs1}) \cdot (\top \wedge \neg\alpha_{mode}(\texttt{macs1}))^* \cdot \texttt{error}(\texttt{macs2}) +$
  $\texttt{error}(\texttt{macs2}) \cdot (\top \wedge \neg\alpha_{mode}(\texttt{macs2}))^* \cdot \texttt{error}(\texttt{macs1}))$

The requirements, specified in section 5.3, can now be expressed in modal logic using the introduced macros. The following requirements are split in two parts: a safety part and a liveness part. The first expresses that *something bad will never happen* and the second expresses that *something good can happen.*

- Deadlock freeness:
  $[\ \top^*\ ]\langle\ \top\ \rangle\ \mathrm{T}$

  This modal formula states that in every state reachable by some sequence of actions at least one action can be performed, i.e. deadlock free.

- Requirement I:
  $[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{drive\}}(i)\ ]\ \mathrm{F}$
  $\wedge$
  $[\ \beta_{drive}\ ]\langle\ \alpha^*_{other} \cdot \mathtt{drive}(i)\ \rangle\ \mathrm{T}$

  with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user \setminus \{start\}} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode}$

  The safety part of the modal formula states that with both control systems in driving mode, it is impossible to go to another mode than driving mode without system errors and without performing user actions (other than `start`). The liveness part states that it is possible to stay in driving mode.

- Requirement II:
  $[\ \beta_{menu} \cdot \alpha^*_{other} \cdot \mathtt{start}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{drive\}}(i)\ ]\ \mathrm{F}$
  $\wedge$
  $[\ \beta_{menu} \cdot \alpha^*_{other} \cdot \mathtt{start}(i)\ ]$
  $\langle\ \alpha^*_{other} \cdot (\ \mathtt{drive}(i) \mid (\alpha_{errorknown} \cdot \alpha_{other} \cdot \mathtt{menu}(i)))\ \rangle\ \mathrm{T}$

  with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode} \wedge \neg\alpha_{errorknown}$

  The safety part of the modal formula states that when a `start` action is given with both control systems in menu mode, it is impossible to go to another mode than driving mode without system errors and without performing user actions. The liveness part states that it is possible to go to driving mode.

  The verified modal formula (see appendix D.2) also expresses that under the same circumstances *both* control systems go to driving mode after a `start` action.

- Requirement III:
  $[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \mathtt{stop}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{menu\}}(i)\ ]\ \mathrm{F}$
  $\wedge$
  $[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \mathtt{stop}(i)\ ]\langle\ \alpha^*_{other} \cdot \mathtt{menu}(i)\ \rangle\ \mathrm{T}$

  with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode}$

  The safety part of the modal formula states that when a `stop` action is given with both control systems in driving mode, it is impossible to go to another mode than menu mode without system errors and without performing user actions. The liveness part states that it is possible to go to menu mode.

The verified modal formula (see appendix D.2) also expresses that under the same circumstances *both* control systems go to menu mode after a `stop` action.

- Requirement IV:

$$[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \texttt{emergency}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{error\}}(i)\ ]\ \mathrm{F}$$
$$\wedge$$
$$[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \texttt{emergency}(i)\ ] \langle\ \alpha^*_{other} \cdot \texttt{error}(i)\ \rangle\ \mathrm{T}$$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user} \wedge \neg\alpha_{mode}$

The safety part of the modal formula states that when a `emergency` action is given with both control systems in driving mode, it is impossible to go to another mode than error mode without performing user actions. The liveness part states that it is possible to go to error mode.

The verified modal formula (see appendix D.2) also expresses that under the same circumstances *both* control systems go to error mode after a `emergency` action.

- Requirement V:

$$[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \alpha_{error}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{error\}}(i)\ ]\ \mathrm{F}$$
$$\wedge$$
$$[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \alpha_{error}(i)\ ] \langle\ \alpha^*_{other} \cdot \texttt{error}(i)\ \rangle\ \mathrm{T}$$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user \setminus \{start\}} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode}$

The safety part of the modal formula states that when a system error occurs with both control systems in driving mode, it is impossible to go to another mode than error mode without performing user actions. The liveness part states that it is possible to go to error mode.

- Requirement VI:

This requirement states that the modes of both control systems have to be synchronized. If one control system goes into another mode, both control systems go into the same mode as fast as reasonably possible. The requirement need to be split in similar modal formulae, one for each mode transition.

$$[\ \top^* \cdot \texttt{menu}(macs1) \cdot$$
$$(\alpha^*_{other} \cdot \alpha_{mode}(macs2))^* \cdot$$
$$\alpha^*_{other} \cdot \texttt{drive}(macs1) \cdot$$
$$(\beta_{outofsync})^n\ ]\ \mathrm{F}$$
$$\wedge$$
$$[\ \top^* \cdot \texttt{menu}(macs1) \cdot$$
$$(\alpha^*_{other} \cdot \alpha_{mode}(macs2))^* \cdot$$
$$\alpha^*_{other} \cdot \texttt{drive}(macs1)\ ]$$
$$\langle\ \beta_{insync} \mid (\alpha^*_{other} \cdot \alpha_{mode} \cdot (\beta_{insync} \mid (\alpha^*_{other} \cdot \alpha_{mode} \cdot (...)))))\ \rangle\ \mathrm{T}$$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{mode}$
$\beta_{outofsync} \equiv (\alpha_{other} \cdot \texttt{menu}(macs1) \cdot \alpha_{other} \cdot \texttt{drive}(macs2) \mid ...)$

$\beta_{insync} \equiv (\alpha_{other} \cdot \texttt{menu}(macs1) \cdot \alpha_{other} \cdot \texttt{menu}(macs2) \mid ...)$

The safety part of the modal formula states that if one control system goes into driving mode (starting in menu mode), it is impossible to stay out of sync for n time samples. The liveness part states that it is possible to synchronize under the assumption that the recursion ends. Note that nothing is said about how fast the mode synchronization is, only that the modes ultimately synchronize. During verification (in section 5.6.4) the performance of the mode synchronization is analyzed.

- Requirement VII:

  This requirement states that during driving mode, a difference between sensor value and setpoint value will result in actuator action in the same time step. The requirement must be split in similar modal formulae, one for each combination of sensor value and setpoint.

  $[\ \beta_{drive} \cdot \alpha^*_{other} \cdot$
  `comm_SensorValue_Get(i,SensorValueL1)` $\cdot$
  `setpoint(i,setpointValueR1)` $\cdot$
  (`actuator(i,actuatorActionL1)`|`actuator(i,actuatorAction0)`) $]$ F
  $\wedge$
  $[\ \beta_{drive} \cdot \alpha^*_{other} \cdot$
  `comm_SensorValue_Get(i,SensorValueL1)` $\cdot$
  `setpoint(i,setpointValueR1)` $]$
  $\langle$ `actuator(i,actuatorActionR1)` $\rangle$ T

  with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user\setminus\{start\}} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode}$

  The safety part of the modal formula states that with both control systems in driving mode, it is impossible to have an actuator action other than turning right, if the computed sensor value indicates -5 degrees and the setpoint is +5 degrees. The liveness part states that it is possible to have an actuator action turning right.

- Requirement VIII:

  This requirement states that during driving mode both actuators are synchronous. The requirement must be split in similar modal formulae, one for each combination of opposing actuator values `actuatorActionR1` and `actuatorActionL1`.

  $[\ \beta_{drive} \cdot$
  $\alpha^*_{other} \cdot$ `drive(macs1)` $\cdot$
  $\alpha^*_{other} \cdot$ `actuator(macs1,actuatorActionR1)` $\cdot$
  $\alpha^*_{other} \cdot$ `drive(macs2)` $\cdot$
  $\alpha^*_{other} \cdot$ `actuator(macs2,actuatorActionL1)` $]$ F
  $\wedge$
  $[\ \beta_{drive} \cdot$
  $\alpha^*_{other} \cdot$ `drive(macs1)` $\cdot$
  $\alpha^*_{other} \cdot$ `actuator(macs1,actuatorActionR1)` $\cdot$
  $\alpha^*_{other} \cdot$ `drive(macs2)` $]$
  $\langle\ \alpha^*_{other} \cdot$

$(\texttt{actuator(i,actuatorActionR1)}|\texttt{actuator(i,actuatorAction0)}) \rangle \mathrm{T}$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user\setminus\{start\}} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode}$

The safety part of the modal formula states that with both control systems in driving mode, it is impossible to have an actuator action on one control system opposing the actuator action on the other control system. The liveness part states that it is possible to have similar actuator actions on both control systems under the same circumstances.

5.6.3 Generation transition system

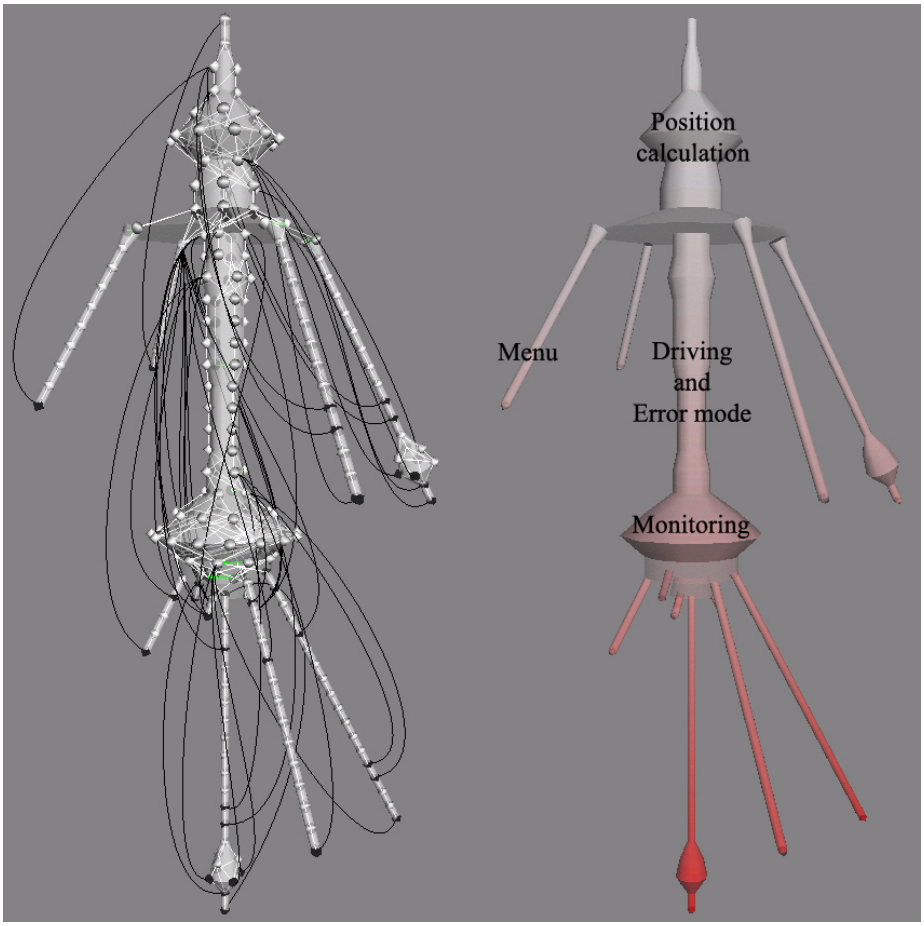

Figure 5.8: Single control system using FSM Visualizer

To verify the control system specification, first the transition system needs to be generated. This is done by using the μCRL toolset (see appendix C.1). First the transition system of a single control system is generated. The transition system is visualized in figure 5.8 using *FSM Visualizer*[3].

[3]3D Visualization tool to analyse transition systems (see appendix C.3)

The FSM Visualizer is also used to analyze the behaviour of the specification in an early stage. With this tool it is possible to interactively explore the transition system in 3D. This has removed errors from the specification and gains an insight into the system. Some subsystems are indicated in figure 5.8.

NoodleView[4] can also be used to analyze the control system's behaviour. However it focuses on state variables and these are only used for the mode switching behaviour. In figure 5.9 the mode switching behaviour is shown. The state variables belonging to `Memory_Mode` and `Memory_Error` are clustered, giving all states in a particular mode with a particular error. Their corresponding transitions are depicted clockwise. Several conclusions can be drawn from this figure. For example, it is impossible to make a transition to driving mode from error mode and it is impossible to make transitions between menu and driving mode if an error has been detected. NoodleView also shows that the controller error is missing in menu mode, since this type of error can only occur in driving mode, when it tries to perform control actions.

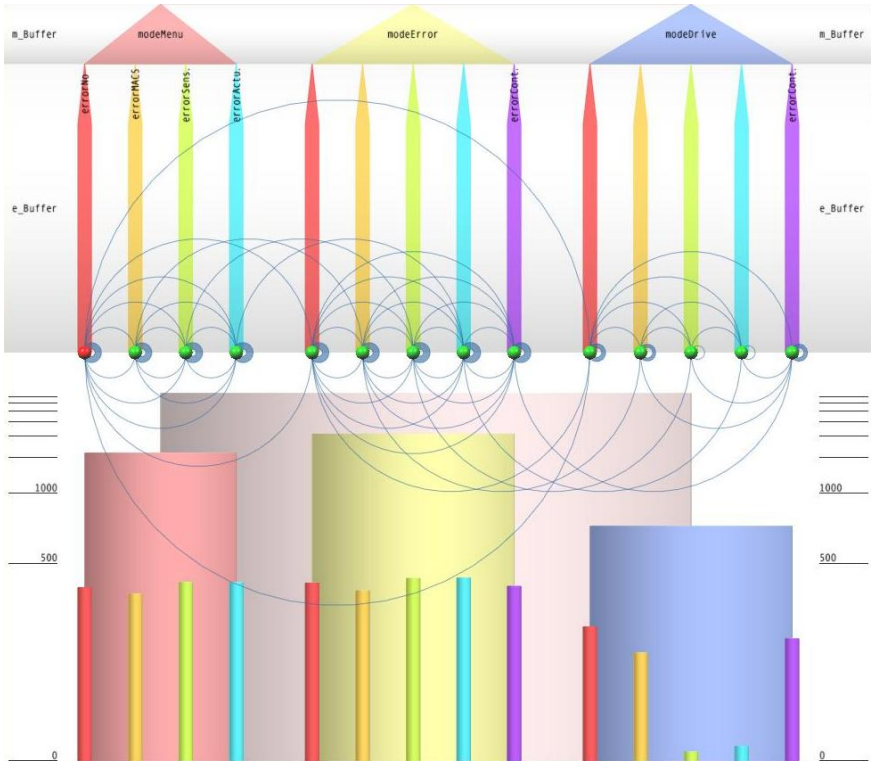

Figure 5.9: Mode switching behaviour in NoodleView

[4]Visualization tool to analyse transition system focused on state variables (see appendix C.3)

Unfortunately the μCRL toolset is not capable in generating the complete transition system at once. By using *Exp.Open*[5] it is possible to generate one transition system from several communicating transition systems. With this tool it is possible to first reduce parts of the transition system, before combining these parts. The resulting transition system will then be a lot smaller compared to generating the complete transition system at once. Due to the introduction of the binary semaphores in section 5.4.3 the complete transition system is reduced by a factor 10 in this particular case. But still it has 1.5 million states, making it impossible to understand the behaviour of the resulting transition system graphically. A detail of the transition system is visualized in figure 5.10 using FSM Visualizer to show the complexity.

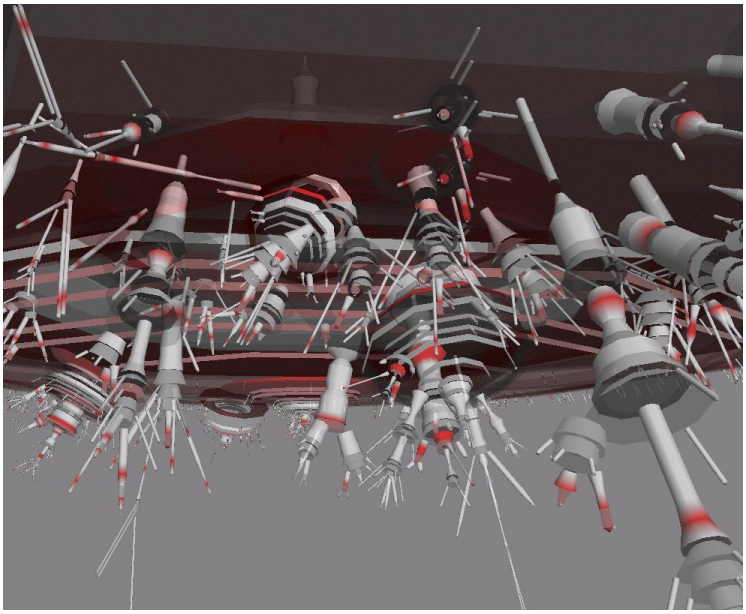

Figure 5.10: Detail of complete control system

NoodleView can also not be used for the complete transition system. The necessary reductions for building the transition system deletes all state variable information, since it maps different states to one state based on trace and branching equivalence.

5.6.4 Results

According to the model-checker *Evaluator*, the specification is deadlock free and the first five requirements are fulfilled. These five requirements are verified with random input i.e. without restrictions on the environment.

[5]Tool in the Cæsar Aldébaran Development Package to generate communicating transition systems (see appendix C.2)

The synchronization requirements however can only be verified with restrictions on the environment. Random and fast alternating input can never be handled properly. The operational modes wil never be synchronized, if one control system only receives `start` signals and the other control system only receives `reset` signals. (Something similar holds for actuator synchronization.) This problem needs to be resolved in the *hardware*. The hardware must guarantee that the input signals are available long enough that both control systems see the input signal and can respond to the input signal. The synchronization requirements can only be verified by building a new transition system with restrictions on the processes `Hardware` and `User`.

To be able to verify mode synchronization, the user actions need to be restricted. The process `User` is only allowed to change the user action once in every 10 time samples. The mode synchronization requirement (requirement VI) can now be fulfilled.

In table 5.1 the worst case synchronization times are shown for the different mode transitions. These values are retrieved by verifying different values n until the requirement is fulfilled.

Mode transition	*Synchronization time*
menu → drive	6 time samples
menu → error	Immediately
drive → menu	4 time samples
drive → error	4 time samples
error → menu	Immediately

Table 5.1: Worst case synchronization times

There are three mode transitions where the synchronization costs time. This is due to the fact that the worst case synchronization is achieved in error mode. One control system makes a mode transition while the other detects an error. The resulting error mode needs to be communicated to the other via canbus and this communication costs time. The transition from menu mode to driving mode is the worst case synchronization overall, since it needs to communicate twice via canbus. This special sequence is shown later in this section.

The other two transitions are synchronized immediately, because these transitions can only be made after user signals and these user signals are also received by the other control system.

The last two requirements could not be fulfilled at the same time, since they are in contradiction. The synchronization requirement is dropped and the other is fulfilled.

5.7 Implementation

The software for the control system is designed in μCRL to be able to validate the software design. However μCRL code can not be compiled to run on a control system directly. This means the μCRL code has to be implemented in another programming environment first. Since it is only possible to use Simulink to generate code for the MACS control system, it needs to be implemented in a Simulink model.

To preserve validity of the software, a formal relation must be made between Simulink and μCRL to map the functionality of the original code to the Simulink model. Unfortunately it is impossible due to the semantical differences between the two programming languages. Simulink is a graphical tool to design and analyze dynamical systems, while μCRL is a process-algebraic programming language to design and analyze communicating processes with data.

The μCRL code can not be simply mapped. To be able to implement the functionality in Simulink, adjustments need to be made. However changes can introduce new faults and can destroy validity. The new model needs to be as close to the original code as possible. Fortunately the model is not changed much to implement the code in Simulink.

One of the differences between the two designs is the treatment of memory and communication. For example memory processes are needed in μCRL to remember values of the previous time sample, but also to control the signal flow between internal processes. In Simulink the values of the previous time sample can be obtained by simply using the predefined `memory` block, which delays a signal by one time sample. The signal flow can be controlled by linking outputs with inputs of different processes. These changes also make the design more transparent in comparison to the μCRL code.

The general composition of the Simulink model is shown in figure 5.11. The major internal processes still exist and the signal flow between internal processes remains the same. First the position is computed from the sensor values, the `supervisor`[6] process computes the mode of the control system, the `controller` process computes the control actions and finally information is sent to the other control system via canbus.

In the μCRL model, the controller is simplified to reduce the complexity of the transition system (to avoid state explosion) without losing the characteristics of a controller. This simplified controller can now be replaced by the two controllers which are developed in chapter 3.

The controller process is shown in figure 5.12. The processes `controlvoter` and `controlswitch` are used to choose the right control strategy: no control, torque control or position control.

Furthermore code is added in the Simulink model to calculate the three posi-

[6]This process also includes monitoring the other control system, unlike the μCRL model.

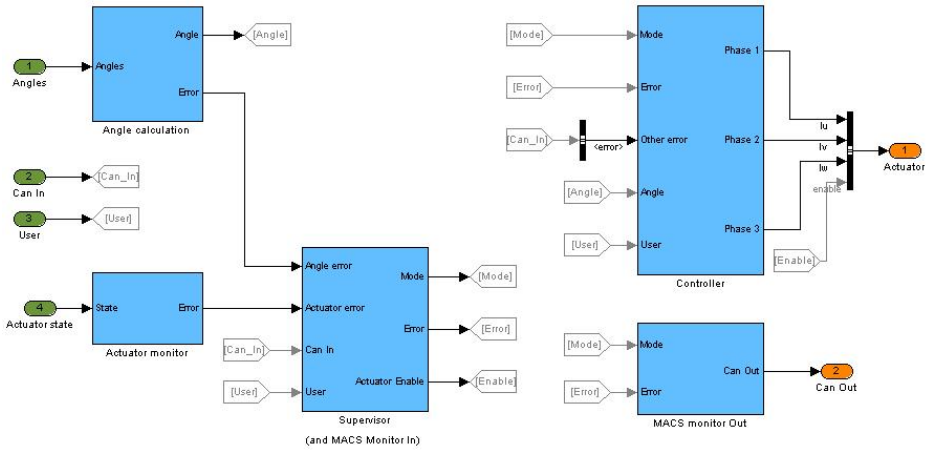

Figure 5.11: Control system model in Simulink

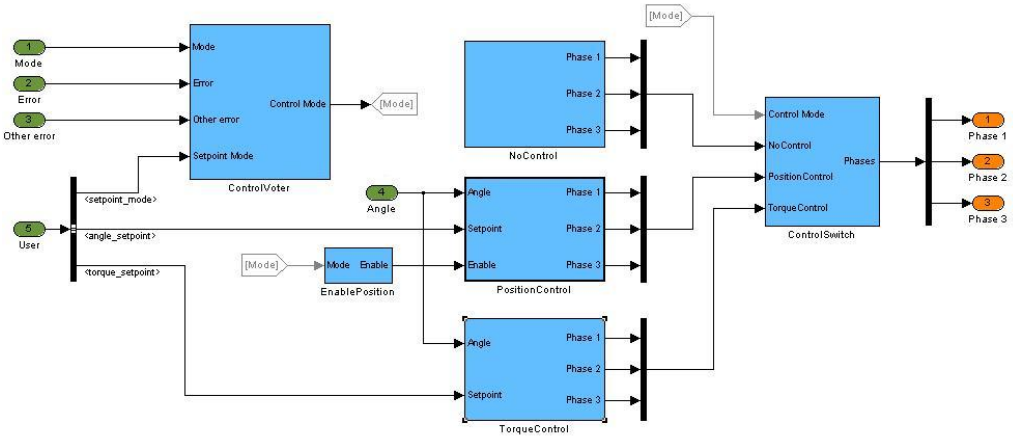

Figure 5.12: Controller process in Simulink

tions from the electrical signals (see appendix A.1) and to be able to send the output and to receive the input.

The Simulink code of the control system did not lead to a single error during testing. Formal methods removed errors in an early stage of the development, making the implementation of code an easy and straightforward task.

Chapter 6

Safety during driving tests

'Simulation is like masturbation: the more you do it, the more you think it is the real thing.'

Maarten Steinbuch

In this chapter the safety during driving tests is analyzed in simulation. Furthermore a strategy is designed to maintain safety during component failures and external emergencies. But first a simulation model needs to be built in Simulink. The model consists of three parts (figure 6.1): the control system (section 6.1), the robot dynamics (section 6.2) and the vehicle model (section 6.3). In this chapter the focus will be on the overall behaviour of the steering robot in representative driving tests.

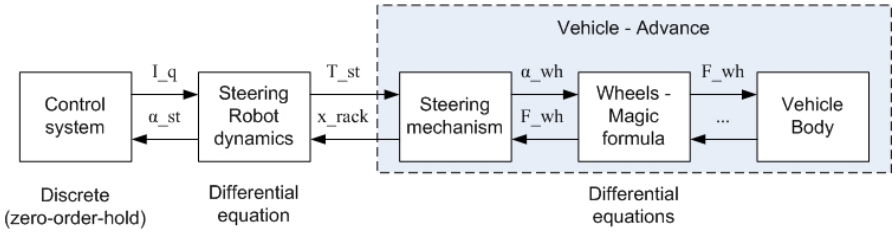

Figure 6.1: Simulation model for steering robot

6.1 Control system

The software for the control system is developed in chapter 5. For the simulation also the behaviour of the control system needs to be implemented. Otherwise the effects of sample frequency and computation delay, described in section 3.2, will not occur.

To realize the effect of sample frequency, the control system is developed as a discrete-time model in a continuous-time environment. The continuous-time signals and operations are discretized. The input signals are converted to discrete signals by using zero-order hold. This means that the input signals are held at fixed values during one time sample. Continuous operations like integrators and differentiators are replaced by their discrete counterparts.

The computation delay could simply be achieved by using a time delay in the output signals. The delay is not exactly known. However the control system runs at the highest possible sample frequency without failing to fulfill all computations and therefore the delay is expected to be half the sample time. Although the delay is random due to differences in computation times, the delay is simulated as a constant delay to reduce complexity of the simulation model.

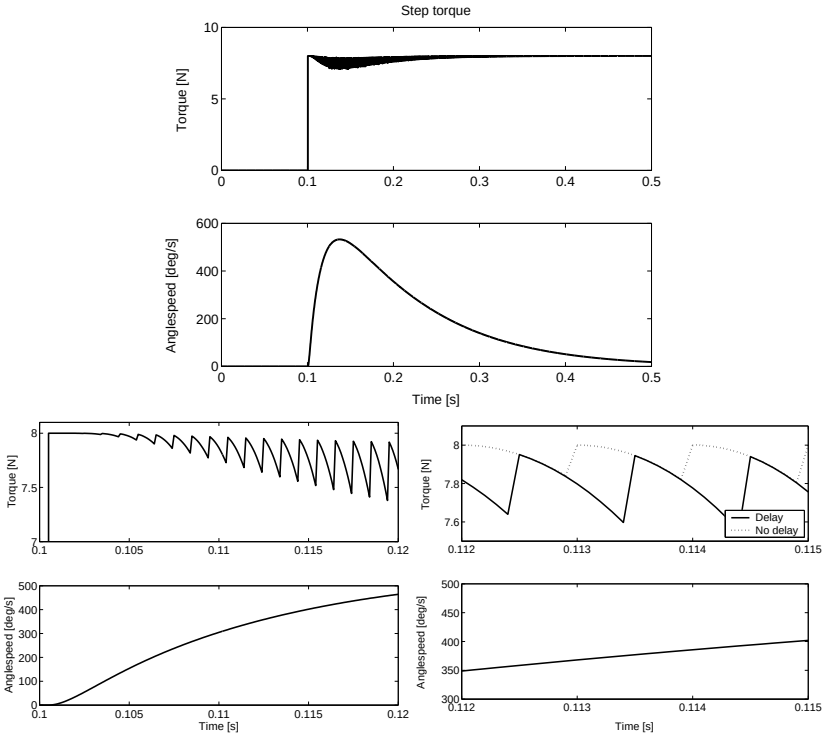

Figure 6.2: Step torque by the control system

The effects of sample frequency and computation delay in the control system model are shown in figure 6.2. In this figure a torque step is computed by the control system and used to manipulate a simple second-order system (mass-spring-damper). The delivered torque[1] is shown in the upper figure. The lower figures are magnifications to show the decrease in delivered torque during one time sample. The lower left figure shows the dependency on angular velocity and the lower right figure shows the computation delay of one half time sample.

[1]The torque is computed from the three phase currents and the actual angle. The computation will be further explained in section 6.2

6.2 Robot dynamics

To be able to make a realistic simulation the dynamics of the steering robot needs to be estimated. A second order system is built with non-linear extensions (friction and backlash) to be able to simulate experiments realistically. The parameters of the dynamics are fitted by real experiments.

6.2.1 Second order estimation

First the dynamic behaviour is represented by a second order system (mass-spring-damper). The dynamic behaviour of the steering robot consists of a mechanical part and an electrical part. It is expected that electrical effects will not play a significant role in the dynamic behaviour, since the mechanical components are much slower. Therefore a realistic fit should be possible by modeling a second order system.

Figure 6.3: The locked steering robot

The parameters of the second order system (inertia J_{gda}, spring constant k_{gda} and damping constant d_{gda}) can be identified experimentally by locking the steering wheel as shown in figure 6.3. Because the steering wheel angle is fixed and the sensor is located in the motor, the sensor measures the angle (α) between steering wheel hub and motor instead of the steering wheel angle. This situation can be modeled as shown in figure 6.4.

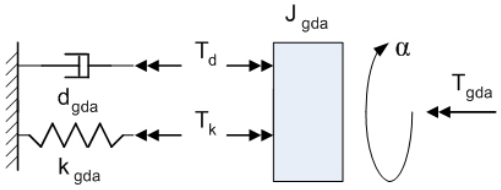

Figure 6.4: Second order system

The resulting differential equation is described as follows.

$$\begin{aligned} J_{gda}\ddot{\alpha} &= T_{gda} - T_d - T_k \\ &\Leftrightarrow \\ J_{gda}\ddot{\alpha} + d_{gda}\dot{\alpha} + k_{gda}\alpha &= T_{gda} \end{aligned} \tag{6.1}$$

6.2.2 Friction

The friction of the system is determined experimentally by measuring the minimal forces needed to change the steering wheel angle. This is done in the same way as the measurement of the torque constant in section 3.2.

The friction forces are fairly high in this setup and the value changes with the steering wheel angle. The steering wheel has preferred positions due to the use of the drive belt. The measured friction coefficient (K_{gda}) is between 0.5 and 1 Nm and is independent of direction.

In this section, the friction force is modeled by coulomb friction only. Viscous friction can be ignored, because it is included in the fitted damping constant. Viscous friction is dependent of speed and can be seen as a damping force. Although it is a rather simple estimation for a complex term like friction, it gives acceptable results without compromising simulation time.

6.2.3 Backlash

Backlash[2] can be measured by applying positive and negative torques to the steering robot, when the steering wheel is locked. The angles give an estimation of the backlash, because after a positive torque the steering robot will be in its positive maximum position and vice versa. The results of the experiments are shown in figure 6.5.

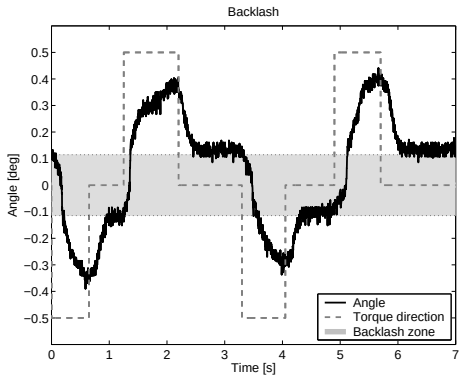

Figure 6.5: Backlash of steering robot

Unfortunately other effects are measured as well, like hysteresis and friction. However the effects are expected to be low. Hysteresis effects can be kept

[2]Backlash is the mechanical term for loss of motion due to clearance between mechanical components. [WIK]

small by applying small forces to the steering robot. The effect of friction is expected to be small, since friction is also small. Backlash ($\alpha_{backlash}$) is therefore estimated at 0.2 degrees.

6.2.4 Fit parameters

The second order model can now be extended with friction and backlash. The model is shown in figure 6.6.

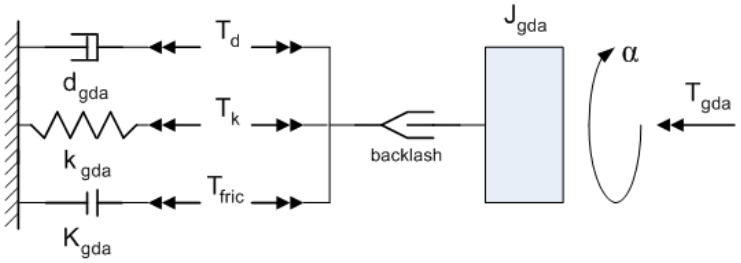

Figure 6.6: Second order system with non linear effects

This model will be used to fit the remaining parameters with a least square fit. Some experiments need to be done to measure the relation between applied torque and motor angles. These experiments can be fitted in the time domain and the frequency domain.

In the frequency domain a transfer function of the steering robot is computed from the relation between torque input and motor angle output. However computing a transfer function is difficult due to the nonlinear effects in the system. Especially the effects of backlash are difficult to model and makes fitting a laborious task.

In the time domain backlash can be avoided by fitting an experiment without backlash. If torque does not change direction (i.e. a step response), backlash plays no role and does not have to be modeled. Only friction has to be added. The differential equation is described as follows.

$$J_{gda}\ddot{\alpha} = T_{gda} - T_d - T_k - T_{fric} \tag{6.2}$$

This experiment can be fitted directly to obtain the remaining parameters of the model.

Although this fit can be used to identify all model parameters, one of the parameters can already be determined without fitting the model. The spring constant (k_{gda}) can be identified by simply measuring the relation between applied torque and relative motor angle. Note that it is only valid if the measurements are done in steady-state.

6.2.5 Model

In the previous sections the parameters of the system are retrieved and these parameters are used to model the dynamics of the steering robot. A new differential equation can be designed, where the steering wheel is no longer secured. The relatively small backlash effect is ignored to avoid simulation problems.

$$J_{gda}\ddot{\alpha}_{gda} = T_{gda} - T_d - T_k - T_{fric}$$
$$\Leftrightarrow$$
$$J_{gda}\ddot{\alpha}_{gda} + d_{gda}(\dot{\alpha}_{gda} - \dot{\alpha}_{st}) + k_{gda}(\alpha_{gda} - \alpha_{st}) + T_{fric} = T_{gda} \tag{6.3}$$
$$\text{with } T_{gda} = i_{st}T_m \text{ and } \alpha_{gda} = \alpha_m/i_{st}$$

The relation between actual steering torque and motor torque (3.6) needs to be adjusted to include the dynamic behaviour of the steering robot. By rewriting (6.3) the actual steering torque can now be described by (6.4).

$$T_{st}(= T_d + T_k) = T_{gda} - J_{gda}\ddot{\alpha}_{gda} - T_{fric} \tag{6.4}$$

The relation between actual steering wheel angle and measured angle (in motor) also needs to be adjusted due to the steering robot dynamics. This relation can be derived from the differential equation, since α_{gda} is the only unknown variable.

The Simulink model can now be implemented. It is shown in figure 6.7.

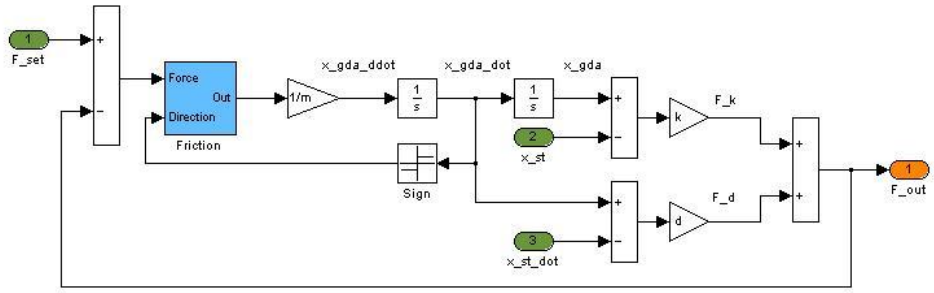

Figure 6.7: Dynamics in Simulink

Ultimately friction is removed from the simulation model to prevent simulation problems. Furthermore the high friction force measured is unacceptable and will be removed in a new design.

6.3 Vehicle model

The last part of the simulation model is the vehicle model. The vehicle model is implemented in Advance, which is a vehicle simulation environment developed

by TNO. First an introduction is given to Advance and in the next sections the vehicle model and steering mechanism is built.

6.3.1 Advance

Advance is a modular vehicle simulation environment, developed at TNO. The tool consists of an extensive library of both vehicle dynamics and powertrain modules, which can be interconnected easily to compose a vehicle model. Advance is developed to be used in the vehicle development process to investigate design effects in simulation.

As mentioned before, the model set up is completely modular. The setup is based on the components which are present in vehicles. This setup allows users to adjust or redesign particular vehicle components using the standard Simulink library. Particular components and control loops can then be tested in simulation. The modular approach also simplifies the setup of hardware-in-the-loop testing, i.e. part of the advance model is replaced by a physical component and tested in a real-time simulation.

6.3.2 Vehicle

The vehicle model can be built from standard Advance blocks, which need to be interconnected. For the simulation a set of parameters is used of an arbitrary mid-sized vehicle. The vehicle model consists of three major parts:

- *Powertrain / Powercontrol*: The powertrain consists of an internal combustion engine, a torque converter, an automatic gearbox and a differential. The differential only drives the front wheels.
- *Chassis*: The chassis consists of four wheels and two roll stabilisers (front and rear). Each wheel has a brake, a spring and a damper. The tyre forces are computed with the Magic Formula-Tyre Model described in [PAC02].
- *Body*: The body behaves as a rigid body, which is subjected to the forces and moments produced by the chassis and powertrain.

6.3.3 Steering mechanism

In Advance no predefined blocks are available to simulate the behaviour of the steering mechanism. The vehicle model only uses wheel angles to control the vehicle direction. The steering mechanism needs to be designed.

The developed steering mechanism is similar to the steering model proposed in [DV03]. The steering model is schematically shown in figure 6.8. It consists of the steering column, the steering rack and power steering unit. The hydraulics of the power steering unit is also modeled, which includes hydraulic delay and a realistic boost curve.

The forces acting upon both ends of the steering rack are the forces generated by the front tyres. The forces are created by two tyre forces: the self-aligning

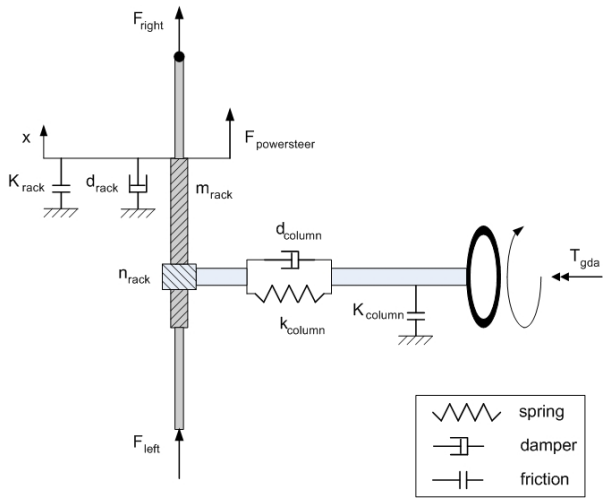

Figure 6.8: Model of steering mechanism

moment and the lateral tyre force. The lateral tyre force does not apply in the center of the wheel and therefore creates a torque around the vertical axis.

The specification of the steering robot is very much influenced by the power steering unit as already mentioned in section 2.5. The efficiency of the power steering unit reduces at higher speeds and ultimately it produces counteracting forces.

The steering model proposed in [DV03] has a realistic model of a power steering unit, but it does not include the steering speed effects. The power steering unit is extended to be able to simulate these effects. It is assumed that the efficiency drops linearly when rotation speeds exceed 360 deg/s and when rotation speeds exceed 720 deg/s the power steering starts producing opposing forces.

In figure 6.9 the effect of power steering on maximum steering speed is shown. Initially the power steering helps to achieve high steering speeds compared to a steering mechanism without power steering. However to realize even higher steering speeds significantly more torque is needed. In this figure the original design proposed in [DV03] is also shown.

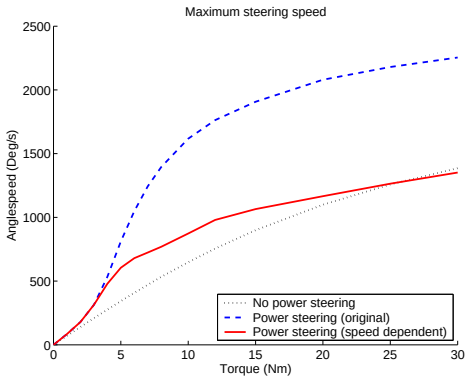

Figure 6.9: The effect of power steering on maximum steering speed

6.4 Simulation

In previous chapters overall safety is realized by validating the software requirements and by duplicating the hardware to make it fault-tolerant. Until now the error mode strategy is not determined. In this section different strategies are investigated to maintain safety during a hardware failure and during external emergencies. However the steering capabilities need to be analyzed first to make sure that the GDA can steer when it is fully operable and more importantly when one of the steering actuators fails.

6.4.1 Steering capabilities

The steering capabilities of the steering robot can best be analyzed using the circle test described in [ISO03]. The steering wheel is instantly set at a constant steering angle, which ultimately leads to a constant steering torque. First the steering capabilities are analyzed without having hardware failures and secondly an analysis is done with failing hardware.

Circle test

In this section the circle test is investigated without having hardware failures. In figure 6.10 a circle test is shown in which the steering wheel angle is instantly set at 20 degrees (top) and 40 degrees (bottom). These values are representative for normal road driving conditions. The actual steering angles and the applied steering torques are shown.

The effect of sample frequency is clearly visible in the applied steering torques. The steering speed is high in the beginning of the circle test to realize the steering angle target, which results in a loss of steering torque as described in section 6.1.

The figure also shows that the static steering torque is relatively low when equilibrium is reached during cornering. The steering torques to maintain these steering angles are around 2 Nm. The high steering torques are only needed to quickly reach the steering angle target.

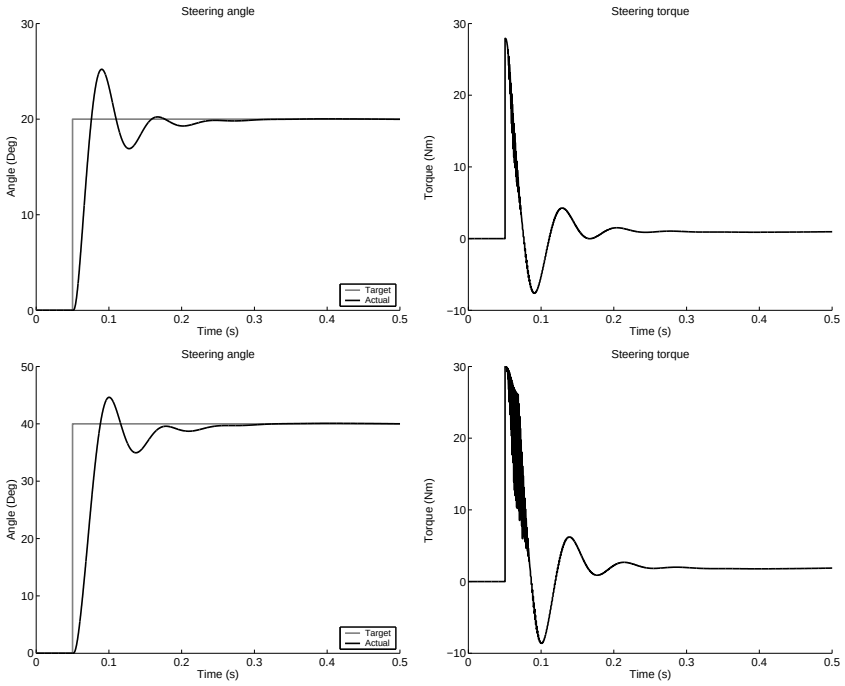

Figure 6.10: Steering angle target of 20 degrees (top) and 40 degrees (bottom)

The resulting trajectory of a circle test with the steering angle instantly set at 40 degrees is shown in figure 6.11.

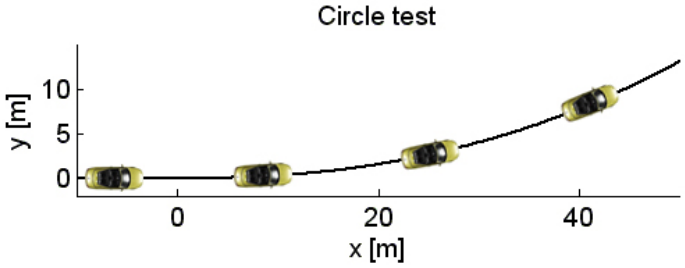

Figure 6.11: Trajectory of circle test

Now it can be verified that during normal driving conditions the steering torques will not exceed 10 Nm, which was concluded in section 2.5 based on low speed road experiments. Simulations are performed in which the steering angle is increased to find the maximum static steering torque during static cornering. The results are shown in figure 6.12 for different speeds.

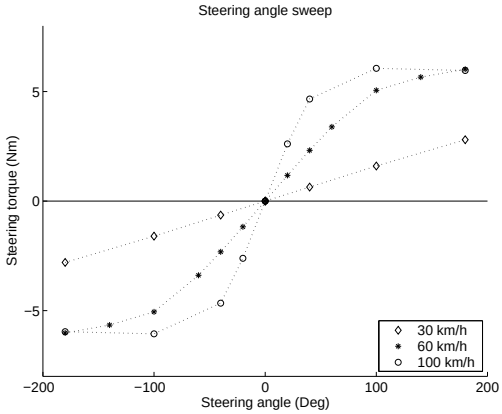

Figure 6.12: Maximum steering torques during static cornering at different speeds

The tyre forces of the circle tests are shown in figure 6.13. The overturning moment M_x and lateral tyre force F_y are directly related to the required torques on the steering wheel as shown in section 6.3. The figures not only show that the overturning moments and overall lateral forces increase if the steering angle increases (until the limit forces of the tyre are reached), but also that the difference between the inner and outer wheel increases. This is due to the fact that the mass is transferred to one side of the vehicle (the roll angle increases) and the vertical tyre forces F_z increase for the outer wheels and decrease for the inner wheels. The simulated tyre forces strongly resemble the typical tyre forces as described in [PAC02].

The results in this section show that 10 Nm will not be needed during normal driving conditions, which was concluded in section 2.5. The static steering torques will not exceed 6 Nm when performing extreme cornering. Based on these torques and the response times it can be concluded that the GDA is capable of steering this vehicle properly.

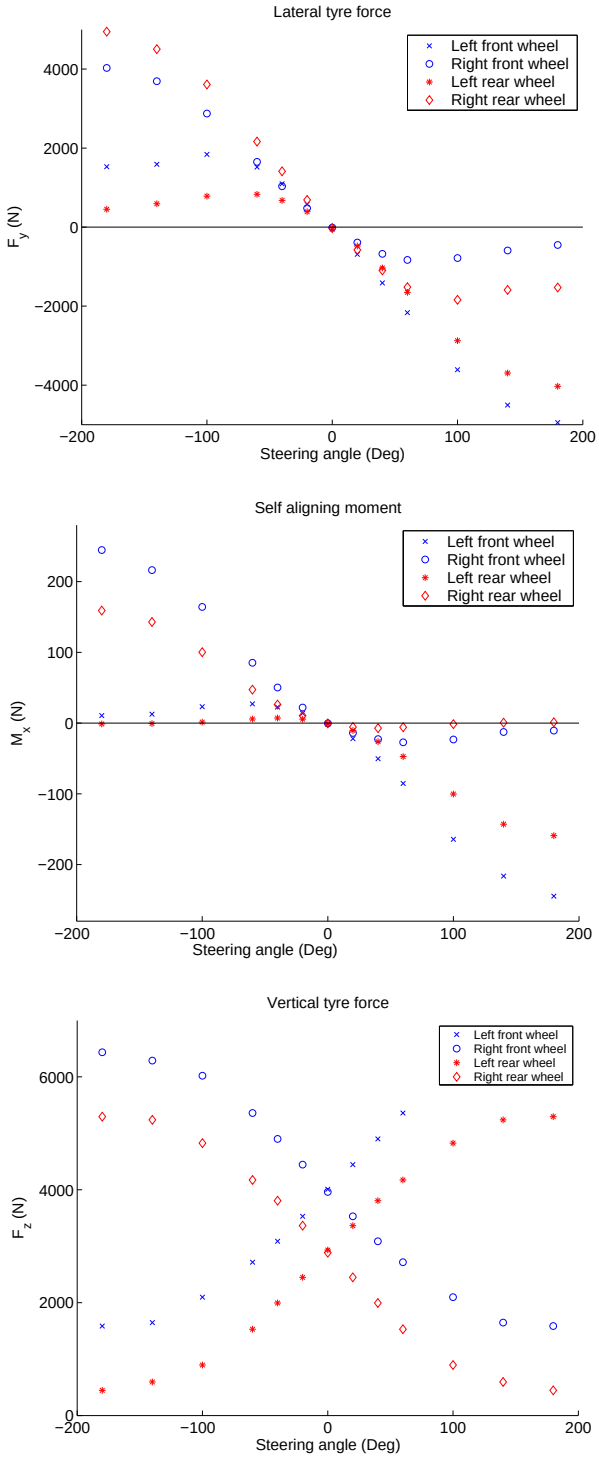

Figure 6.13: Tyre forces during static cornering at 60 km/h

Hardware failure during a circle test

The most critical hardware failures are failures which result in a loss of one of the two actuators. This means that only half of the maximum steering torque can be used. The loss of one actuator in the current setup can be caused by a hardware failure of a control system, a servo amplifier or the actuator itself. The effect of this loss is analyzed in this section.

First of all a failure needs to be detected and this will take time. Failure of an actuator and the servo amplifier can be detected by the onboard error detection functionality of the servo amplifier (see section 4.3). The control systems monitor each other by using alive signals (see chapter 5). If a hardware failure is detected, the controller needs to be adjusted to compensate for the loss of an actuator. The control system will double its control effort within the capabilities of the remaining hardware.

The hardware failure detection needs to be investigated. In figure 6.14 the effect of an undetected hardware failure is shown. During a circle test one actuator is lost and the other controller still proceeds with the original algorithm, which is no longer adequate. In this particular circle test the steering angle drops by 5 degrees and it takes almost 0.5 seconds to reach the target again.

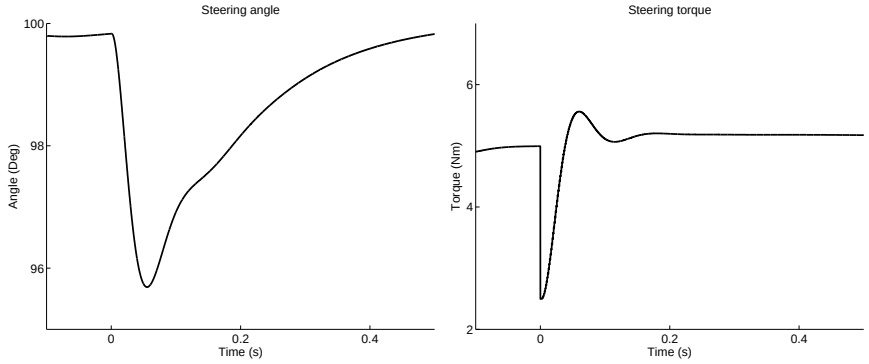

Figure 6.14: Undetected hardware failure

Figure 6.15 shows what happens if the hardware failure is detected and the controller is adjusted. For a short period the controller is still unaware of the loss of the other actuator and the applied steering torque is only half of the required steering torque. During the detection time the steering angle will deviate from the target value. When the controller is aware of the hardware failure, it will double its control effort. Unfortunately this control effort is limited by the maximum steering torque of a single actuator.

The monitoring mechanism is expected to be the slowest fault detection mechanism and therefore the most critical. Based on the simulations of chapter 5 the monitoring mechanism takes roughly 5 time samples to conclude a control system failure. Therefore detection times are simulated of 5 milliseconds and 10 milliseconds. In this circle test the steering angle drops by at most 1 degree

and it takes less than 0.2 seconds to reach the target again.

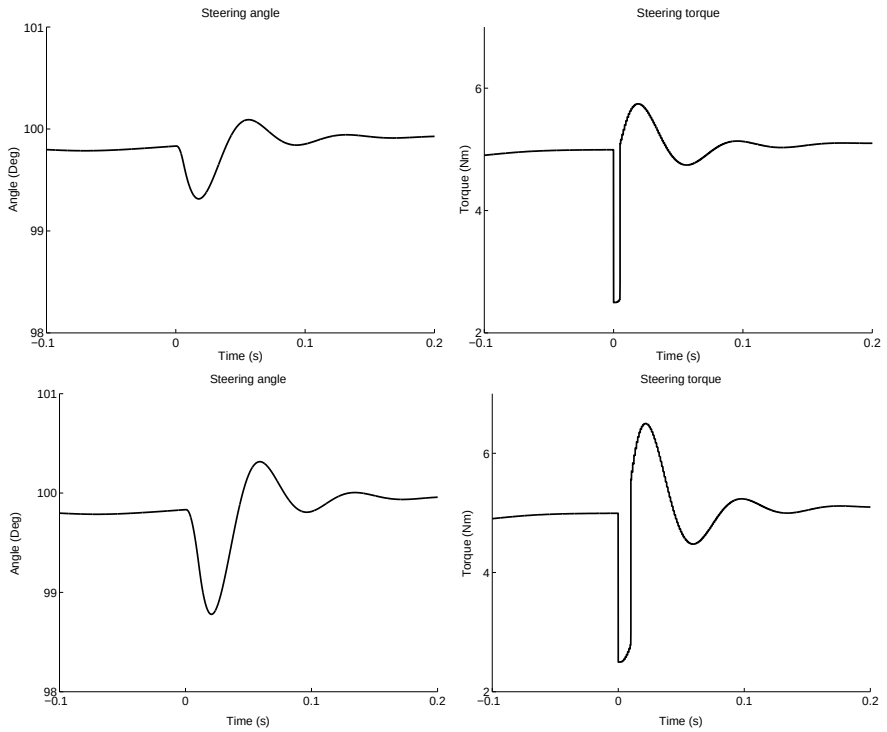

Figure 6.15: Hardware failure detected in 5 time samples (top) and 10 time samples (bottom)

Obviously the reduced maximal steering torque will also effect the circle test. However it will only effect the settling time, because the steering torques of a single actuator is still more than enough to achieve reasonable steering torques for the static behaviour. The settling time decreases, because the maximum steering speeds decreases.

Figure 6.16 shows the effect of a hardware failure on the maximum steering speeds. With two actuators functioning steering speeds of 1400 deg/s can be achieved, when only one actuator can be used, maximum steering torque drops and the maximal steering speed is reduced to 875 deg/s. The steering speeds achieved by one actuator are relatively high compared to two actuators. This is caused by the behaviour of the power steering at high steering speeds, which is described in section 6.3. Note that the effect of sample frequency (described in section 6.3) is clearly visible and is extremely high.

Based on the results of the circle test, it must still be possible to control the vehicle with only one actuator. Although high steering speeds can no longer be achieved, which would limit driving tests with high demands on steering, one actuator can still produce enough torque to steer the vehicle under normal driving conditions. For example, to stop the vehicle safely in error mode.

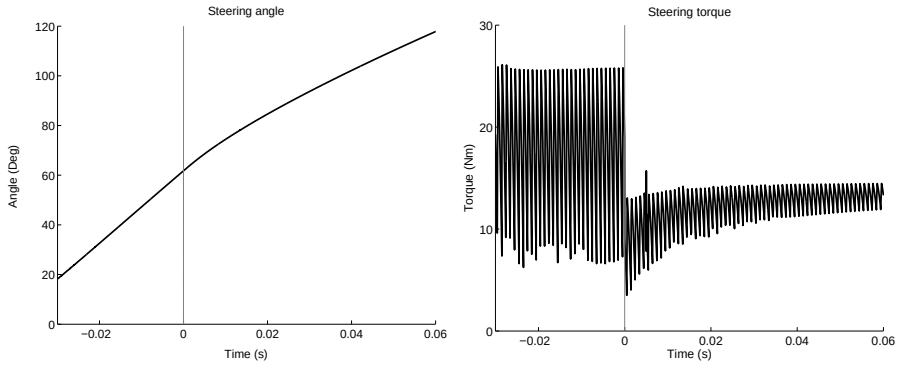

Figure 6.16: Effect of a hardware failure on maximum steering speeds

Note that one actuator can only steer the vehicle as long as the steering torques can be produced by one actuator. In this circle test a mid-sized vehicle is used for which a single actuator can produce enough steering torque. If for example the vehicle mass is doubled, the tyre forces will increase and as a result one actuator can no longer produce the required steering torque under all circumstances during the circle test.

6.4.2 Error mode strategy

So far the best way to control the vehicle in error mode is not investigated. It is not yet clear, whether the GDA should brake or attempt to continue the prescribed track in case of hardware failure or emergency. First a suitable driving test is determined to analyze the dynamic behaviour of the GDA. This driving test is then used to determine the error mode strategy for failing hardware and when an emergency occurs.

Double lane change test

The effect of failing hardware is best analyzed by using driving tests with high demands on steering speed. The double lane change is one of the most demanding driving tests, because the vehicle is pushed to the limit to drive through the desired trajectory at the highest possible speed. For this analysis, the ISO 3888 [ISO99] double lane change driving test is chosen. Angle setpoints are determined experimentally, which let the vehicle drive through the desired trajectory at its highest possible speed. The maximum speed for the trajectory is determined at 60 km/h. The actual steering angles and steering torques are shown in figure 6.17.

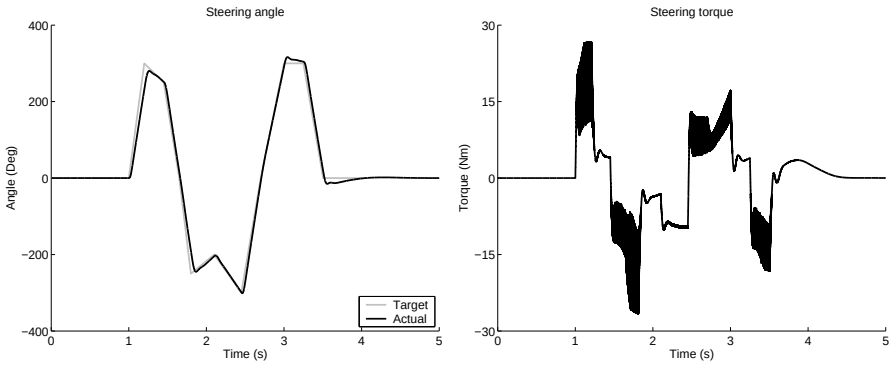

Figure 6.17: Double lane change at 60 km/h

Figure 6.18 shows the resulting double lane change trajectory with normal functioning control. The vehicle drives along the desired trajectory without hitting the traffic cones.

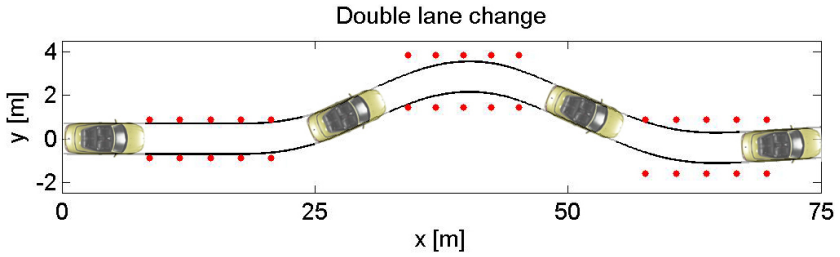

Figure 6.18: Trajectory of double lane change at 60 km/h

Hardware failure strategy

Based on the double lane change test and the circle test described earlier a strategy needs to be designed during a hardware failure. If one actuator is lost due to a hardware failure, the other actuator tries to double its efforts within its specification. Unfortunately some performance is lost.

The actual steering angles and steering torques are shown in figure 6.19 when the double lane change test is continued with a single actuator and the same steering setpoints. After 1.4 seconds an actuator failure occurs and the maximum steering torque is reduced.

The actual steering angles do not deviate much compared to the situation with both actuators available. The high torques are only needed for maximal steering speed, which is already shown by the circle test. Fortunately the difference in steering speed is not that high, because the extra steering torque is partly lost

to the power steering (described in section 6.3).

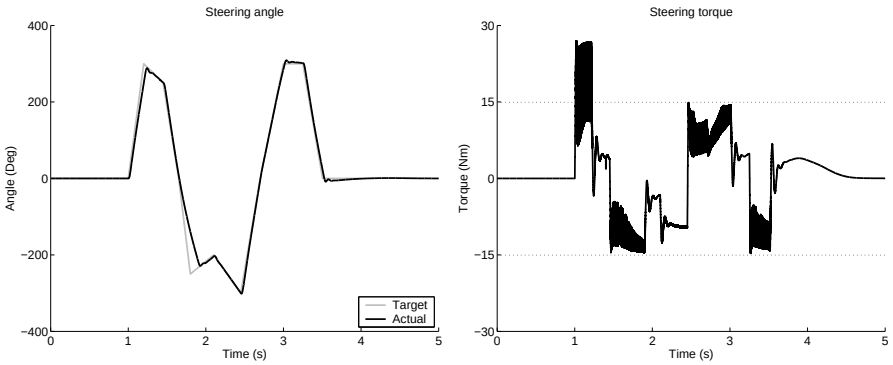

Figure 6.19: Double lane change at 60 km/h: Single actuator after 1.4 s

Figure 6.20 shows the resulting trajectory of the double lane change driving test. Although the prescribed trajectory can no longer be fulfilled with a single actuator, the followed trajectory is acceptable from a safety point of view.

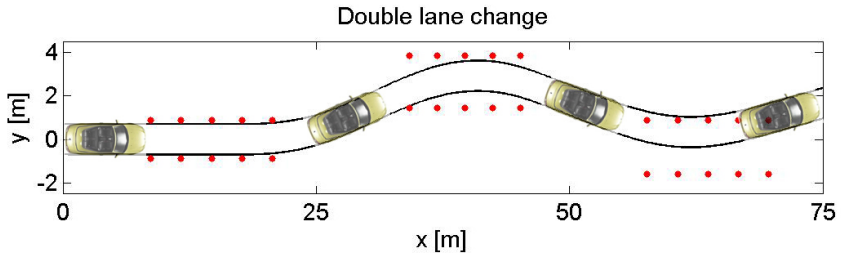

Figure 6.20: Trajectory of double lane change at 60 km/h: Single actuator after 1.4 s

It is also an option to stop the vehicle if a hardware component fails. However chances are that the vehicle gets into an slip, which can not be controlled. During a double lane change driving test the vehicle momentarily starts drifting, which makes braking a risky strategy. The effect of braking is shown in figure 6.21. The vehicle clearly gets out of control.

Another strategy to stop the vehicle is to steer the vehicle in a straight line before braking. However it is not possible to steer the vehicle in a straight line without feedback of vehicle parameters like slip angle, yaw rate, etc. Simulation showed that simply setting the steering angle to zero degrees or decreasing the steering torque leads to a similar trajectory as the trajectory shown in figure 6.21.

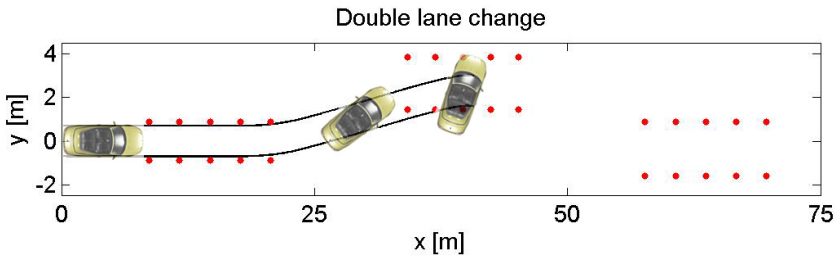

Figure 6.21: Double lane change at 60 km/h: Braking after 1.4 s

Based on the results of the double lane change and the circle test, the best open-loop strategy during hardware failure is to try to complete the trajectory with the remaining hardware. Performance will degrade, but it is better to deviate slightly from the desired trajectory than to cause the vehicle to get out of control. Even driving tests with high demands on steering speeds will not deviate much from the desired trajectory.

To reduce the risk of complete loss of control (two subsequent hardware failures), the driver needs to be informed of the hardware failure and the GDA is prevented from further use until the hardware is replaced or fixed.

Although comparable vehicles may behave differently during a double lane change test, the effect on its behaviour during a hardware failure will be similar. Other mid-sized vehicles (and smaller vehicles) will need similar (or smaller) steering torques, which can also be produced by a single actuator. Furthermore the steering robot will also have short periods in which the required steering speeds can no longer be achieved. However this will not effect the steering characteristics much either and the same conclusions can be drawn for similar vehicles. Therefore the behaviour of other mid-sized (and smaller) vehicles during the double lane change is not discussed here.

Emergency strategy

The strategy during an emergency also needs to be designed. Following the desired trajectory is not an option. If something goes wrong due to external factors, the driving test has to stop at once. One can think of people or other vehicles which are on the track accidentally.

Two situations can be distinguished: driving tests with and driving tests without a test driver. A test driver can take over the vehicle in case of an emergency. The GDA is simply shut down, when an emergency signal is given by the test driver.

If something goes wrong during an autonomous driving test, the vehicle needs to stop directly. Without further information, the best strategy to achieve this is by full braking. The fact that the vehicle can get in a slip is inevitable.

Safe test area

With the error mode designed, it is possible to predict which area is needed for a particular driving test regardless of possible hardware failures and emergencies. In other words safety can be guaranteed as long as the test platform is big enough and if people stay outside this area.

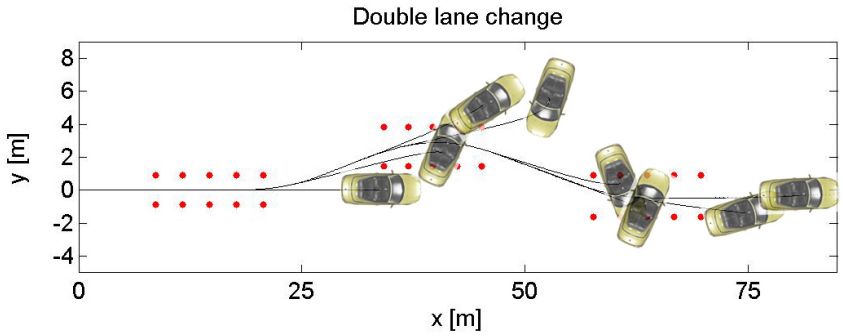

Figure 6.22: Area needed for double lane change at 60 km/h

The driving test is simulated with emergencies at different times in figure 6.22. The figure only gives an indication for the area needed for a double lane change driving test for this particular speed and vehicle. Other vehicles may drive at a higher speed and behave differently at full braking, which results in another area.

Chapter 7

Conclusions & Recommendations

'Reality is both continuous and discrete.'

Geoff Haselhurst

7.1 Conclusions

The conclusions are split up in three parts. Conclusions are given on controller design, safety-critical design and safety during driving tests.

7.1.1 Controller design

For the development of the GDA, first the steering robot needs to be operational. The development of the steering robot has been disappointing. Looking back the project has been too ambitious. The steering robot design has too many uncertain factors and experimental components. Some of the problems are stated below:

- Sine generation principle does not work
- The original control system (MACS 555) is not powerful enough. Subsequently a more powerful prototype is used (MACS 565). Unfortunately it has had failing IO and some problems have occurred with its code generation. These problems are caused by the fact that it is still under development during this project.
- Off-the-shelf servo-amplifiers are not performing well enough. The output currents are generally too low for low-voltage amplifiers available.
- A beta-prototype with improved hardware layout is designed but due to time and budget restrictions, it isn't manufactured during this project.

- The motor output is much less than specified.

The steering robot has been operational albeit with reduced performance and with only one motor. The torque controller is tested in reality. The position controller however is only tested in simulation due to the problems with the hardware.

7.1.2 Safety-critical design

The safety of the steering robot is improved by using redundant hardware. If one hardware component fails, at least one other component is functioning and the steering robot can still produce torque. It is assumed that the produced torque is large enough to outperform the failing component

Using spare actuators is not an option due to packaging vs. performance issues. To maintain control two actuators are used simultaneously. This way at least one actuator can try to prevent unsafe situations.

Two hardware designs are proposed to improve safety. The first hardware design compromises software safety, because it introduces a need for synchronization. However the other hardware design needs difficult additional hardware components. Ultimately the dual control setup is chosen for the control system, where each actuator has its own controller. This setup reduces hardware complexity, but increases software complexity due to the synchronization problems.

By looking at both hardware and software aspects in the design process (so-called *hardware/software codesign*), a better understanding of all possible options is obtained, which ultimately leads to a better decision.

The development of the control system software has had little problems. The use of (classical) formal methods certainly proves to be a good way to manage the complexity even in this hybrid environment. The requirements of the control system can be verified in μCRL and the implementation of the model in Simulink works without any problems.

During the verification of the model problems with the original software design have been found, which are almost impossible to detect without formal methods. Sequences of user actions have been discovered, which conflicted with the requirements. Furthermore conflicting requirements have been found. Although the basic functionality of the control system is fairly simple, the hardware redundancy introduces a lot of synchronization problems due to the parallelism.

By using redundancy there is definitely a trade-off between hardware safety and software safety. Additional hardware components are used to improve hardware safety, but makes the synchronization a lot more complex, which can deteriorate software safety. In this case the GDA becomes safer, because the safety of the software is verified. However care must be taken when hardware redundancy is introduced. It is not unimaginable that hardware redundancy makes the complete system even less safe.

The formal languages (like μCRL) are useless without good tools to support them. The μCRL toolset contains powerful tools to generate transition systems, but it needs to become more user-friendly. For example by making a user interface, where you can choose between operations and options. The visualization tools (FSM Visualizer and NoodleView) proved to be good tools to get insight in the models. CADP is used to generate the GDA transition systems and to verify the requirements using modal logic.

The transition system of the GDA is too large for the tools and some tricks are used to reduce its size. First parallelism is reduced by grouping actions in critical sections. Secondly the transition system is significantly reduced by reducing parts of the transition system before combining them.

7.1.3 Safety during driving tests

Since the GDA is not operational on time, driving tests can only be done with simulations. A simulation model is built in Simulink to analyze the behaviour of the GDA during driving tests. The model includes the control system, the steering robot dynamics and a vehicle model. The safety during driving tests can not be verified formally, simulations are used instead.

The behaviour is analyzed using a circle test and a double lane change test. Simulations show that if a hardware failure occurs, one actuator can still perform reasonably. Two actuators are only needed to oppose the counteracting moments of the power steering unit for extreme steering speeds. Therefore static steering angles can still be maintained and the maximum steering speed only drops slightly.

The simulation model is also used to find out what the best strategy is for the error mode. Based on the results of simulations with the circle test and the double lane change driving test, the best strategy during hardware failure turned out to be to complete the trajectory with reduced capabilities. To design a better strategy to stop the vehicle, feedback on particular vehicle parameters would be needed. The best strategy during an external emergency turned out to be to switch off the GDA if a test driver is in the vehicle and to brake immediately in case of an autonomous driving test.

7.2 Recommendations

The major recommendation of this thesis is getting the GDA operational. The hardware problems with the GDA's steering robot need to be resolved:

- *Direct drive design*
  The new direct drive design needs to be built to replace the current steering robot with drive belt. This will eliminate the friction problem and the new motors will produce sufficient power. Due to the direct drive design the steering ratio is significantly reduced, which removes the sample frequency problems.

- *Servo amplier*
  A new and more powerful servo amplifier is built to replace the off-the-shelf product. The final problems with the electronics need to be resolved.

With the GDA operational, it is possible to validate the driving simulations done during this project. The submodels can also be improved. Now some parameters are estimated (e.g. steering robot dynamics), because they simply couldn't be retrieved without an operational GDA.

Furthermore the hardware safety can be tested by making components fail deliberately, i.e. *fault seeding*.

The next phase in the development of the GDA should be towards a path following robot. This way the behaviour during handling tests will be more representative, and we expect that the added feedback possibilities will improve the safety in error mode. This can be achieved by extending the GDA with a GPS unit and use its output to correct the followed path.

In the future it should be possible to model both the continuous and discrete behaviour of the GDA by using hybrid languages like χ [BR00], HyPA [CUI04] and hybrid automata [HEN96]. The advantage of such combined languages is that we do not need to worry whether the decoupling between software and hardware in our models is justified. However, the current tools for such languages do not support models of the desired complexity.

Chapter 8

Discussion

'There are no hybrid systems, there are only hybrid models.'

Peter Struss

During my graduate studies I noticed how a lot of research on the university is too narrowly focussed on one particular field, neglecting to consider (partial)solutions in other areas. Unfortunately many problems are not easy to put in one particular research area.

I truly believe that disciplines should be mixed in an early design phase to get a better understanding of each other. This means that problems should be tackled by interdisciplinary teams more often. It is a common misconception that supporting software can quickly be developed as a final step in the process instead of creating hardware and software in parallel (which ensures better compatibility).

The problem is that the complexity of software and the developing time is hugely underestimated in many cases by non-computer scientists (and even by computer scientists). This leads to project delays or even to failing projects. Engineers can no longer explain why their software is malfunctioning under particular circumstances due to the complexity of the software.

Fortunately a lot of research is done on formal methods to get an insight in the complexity. Hybrid languages are being developed to be able to simulate and verify the discrete and continuous behaviour: e.g. χ [BR00], HyPA [CUI04] or hybrid automata [HEN96]. This makes it possible to design a single model with both discrete and continuous elements (e.g. software and hardware) and verify their combined requirements.

In this master's thesis an artificial split is made between discrete and continuous behaviour to be able to verify the requirements individually. μCRL is used to verify the discrete behaviour and Simulink is used to simulate the continuous

behaviour. By using a hybrid language this split could be avoided. However hybrid languages are still in its infancy, verification of hybrid models is not possible yet and the simulation model of the GDA is too large for current simulation tools.

Bibliography

[AL90] Anderson, T., Lee, P.A., *Fault Tolerance: Principles and practice*, Springer-Verlag, second edition, ISBN 3-211-82077-9, (1990)

[ABD04] Anthony Best Dynamics, *Description and specification - Steering robot SR30-Lite / SR30-Omni*, Anthony Best Dynamics, (2004)

[AMN86] Arkenbosch, M., Mom, G., Nieuwland, J., *De nieuwe Steinbuch - Het rijdende gedeelte*, Kluwer Technische Boeken, ISBN 90-201-1972-9, (1986)

[ATI97] ATI/Heitz, *The ATI programmable steering machine*, ATI/Heitz, (1997)

[AVI85] Avižienis, A., *The N-version approach to fault-tolerant software*, Published in *IEEE Trans. Software Eng., Volume 11* pp. 1491-1501, IEEE, (1985)

[BR00] Beek, D.A. van, Rooda, J.A. *Languages and applications in hybrid modelling and simulation: Positioning of Chi*, Published in *Control Engineering Practice vol. 8, nr. 1* pp. 81-91, Elsevier, (2000)

[BS01] Bradfield, J.C., Stirling, C., *Model logics and mu-calculi*, Published in *Handbook of Process Algebra* pp. 293-330, Elsevier, (2001)

[BV04] Braig, M., Verschuren, R.M.A.F., *Development of a Simulink Model of a Brushless AC Motor*, TNO, TNO Report 04.OR.AC.022.1/RMV, (2004)

[BW90] Baeten, J.C.M., Weijland, W.P., *Process Algebra*, Cambridge University Press, ISBN 0-521-40043-0, (1990)

[CES86] Clarke, E.M., Emerson, E.A., Sistla, A.P., *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*, Published in *ACM Transanctions on Programming Languages and Systems 8* pp. 244-267, ACM, (1986)

[CUI04] Cuijpers, P.J.L., *Hybrid Process Algebra*, Eindhoven University Press, ISBN 90-386-0972-8, (2004)

[DV03] Duringhof, H.M., Verschuren, R.M.A.F., *Confidential report*, TNO, TNO Confidential Report, (2003)

[DV04] Duringhof, H.M., Vermeer, E., *Brushless AC Motor for Steer-By-Wire application*, TNO, TNO Report 04.OR.AC.042.1/HMD, (2004)

[DYK65] Dijkstra, E.W., *Cooperating sequential processes*, Reprinted in "Programming languages" (1968), Academic Press, (1965)

[FGR05] Fokkink, W.J., Groote, J.F., Reniers, M.A., *Modelling Distributed Systems*, Technical University of Eindhoven, Draft version, (2005)

[FPE94] Franklin, G. F., Powell, J.D., Emami-Naeini, A., *Feedback control of dynamic systems*, Addison-Wesley Publishing Company, Inc., third edition, ISBN 0-201-53487-8, (1994)

[GM98] Groote, J.F., Mateescu, R., *Verification of temporal properties of processes in a setting with data*, Centrum voor Wiskunde en Informatica, SEN-R9804, (1998)

[GPW01] Groote, J.F., Pang, J., Wouters, A.G., *Analysis of a Distributed System for Lifing Trucks*, Centrum voor Wiskunde en Informatica, SEN-R0111, (2001)

[GP95] Groote, J.F., Ponse, A., *The syntax and semantics of μCRL*, Published in *Algebra of Communicating Processes '94* pp. 26-62, Springer-Verlag, (1995)

[HEN96] Henzinger, T.A., *The theory of hybrid automata*, Published in *11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)* pp. 278, IEEE, (1996)

[ISO99] ISO, *Passenger cars - Test track for a severe lane-change manoeuvre* , ISO, ISO 3888, (1999)

[ISO03] ISO, *Road vehicles - Lateral transient response test methods - Open-loop test procedure*, ISO, ISO 7401, (2003)

[LEW96] Lewis, E.E., *Introduction to Reliability Engineering*, Wiley, second edition, ISBN 0471-01833-3, (1996)

[MS00] Mateescu, R., Sighireanu, M., *Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus*, INRIA, INRIA Research Report RR-3899, (2000)

[MSS99] Müller-Olm, M., Schmidt, D., Steffen, B., *Model-Checking - A Tutorial Introduction*, Published in *Lecture notes in Computer Science, Volume 1694* pp. 330-354, Springer-Verlag, (1999)

[PAC02] Pacejka, H., *Tyre and vehicle dynamics*, Butterworth-Heinemann, first edition, ISBN 0-7506-5141-5, (2002)

[SAE96] SAE, *Steady-state directional control test procedures for passenger cars and light trucks*, SAE, SAE J266, (1996)

[STO96] Storey, N., *Safety critical computer systems*, Addison-Wesley, ISBN 0-201-42787-7, (1996)

[STO99] Storey, N., *Design for Safety*, Published in *Towards System Safety: Proc. 7th Safety-Critical Systems Symposium* pp. 1-25, (1999)

[WOU01] Wouters, A.G., *Manual for the μCRL tool set*, Centrum voor Wiskunde en Informatica, SEN-R0130, (2001)

[ABD] Anthony Best Dynamics, *http://www.abd.uk.com*, (2005)

[ATI] ATI/Heitz, *http://www.atiheitz.com*, (2005)

[HOR] Horiba Instruments, *http://www.emd.horiba.com*, (2005)

[STA] Stähle, *http://www.stahle.com*, (2005)

[WIK] Wikipedia, *http://en.wikipedia.org*, (2007)

Appendix A

Hardware

In this chapter the angle measurement and torque measurement is further explained.

A.1 Angle measurement with the Netzer encoders

In the first section the Netzer encoder itself is shortly explained. In the second section the algorithm is given how to compute the electrical angle from the Netzer signals.

A.1.1 Netzer encoder

The Netzer electrical encoders give sine and cosine signals, which provide a unique combination to identify a mechanical angle. It is shown in figure A.1.

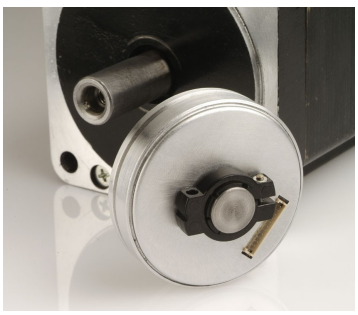

Figure A.1: Netzer electrical encoder

It can be operated in two modes: coarse mode and fine mode. The coarse mode can be used to find the mechanical angle during initialization, because it has only one electrical cycle per revolution. After the initialization phase the fine mode can be used. The fine mode has several electrical cycles per revolution, making it possible to calculate the mechanical angle more precisely.

The Netzer encoder has one input and three output signals. The input signal is a signal to select fine or coarse mode. Two of the output signals are the sine and

cosine signals. The sine and cosine signals of the Netzer encoder are relative to the third output: reference voltage signal. The next section explains how to compute the electrical angle from the sine and cosine signals.

A.1.2 Electrical angle computation

To be able to compute the mechanical angle an algorithm needs to be designed to compute an electrical angle from the sine and cosine signals.

The implemented algorithm works as follows:

- Compare the signs of the sine and cosine signals, and identify the corresponding quadrant.
- Compare the absolute values of the sine and cosine signals, and identify the corresponding octant.
- Divide the smaller absolute value by the larger one to determine the tangent of the electrical angle, shifted into the first octant.
- The arctangent provides the electrical angle, shifted into the first octant.
- Relocate the shifted angle to its original octant.

A.2 Torque measurement

The torque is measured with strain gauges. Due to the applied torque, the length of an individual strain gauge changes, resulting in a change in electrical resistance. This resistance change can then be measured and used to predict torque.

In the steering robot four strain gauges are placed in a Wheatstone bridge configuration, which is shown in figure A.2. A Wheatstone bridge consists of two parallel voltage dividers. One voltage divider consists of R_1 and R_3; the other consists of R_2 and R_4. These four strain gauges are identical, making it a balanced bridge. If a torque is applied, two strain gauges laying opposite will compress and the other two will stretch. As a result, the output voltage V_{out} will change. If the proper materials are used for the strain gauges, the voltage is linear with the applied torque in a certain range.

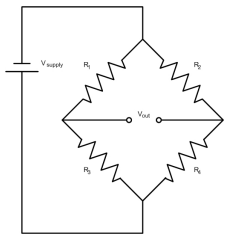

Figure A.2: Strain gauges in Wheatstone bridge

Appendix B

Transition system reduction using binary semaphores

B.1 Introduction

In many cases the transition system is too large to be generated due to the *state explosion* problem. The transition systems need to be reduced. Mostly this is done by hiding the majority of the actions. Smart techniques are then used to find smaller branching bisimilar transition systems. However a lot of information is lost, making it very hard to find the causes of errors in the specification.

An important cause for state explosion is the use of parallel processes in the specification, k parallel processes with n_i transitions have in the worst case $\prod_{i=1}^{k}(n_i + 1)$ combined states. Parallel processes will quickly generate more states than computers can handle. The number of states caused by parallelism can be reduced by using binary semaphores in the specification.

B.2 Technique

This new technique is based on the fact that particular parts of the processes are independent and can be treated as such. If these parts are within a parallel composition, their behaviour will not be effected by actions of other processes. Therefore it is allowed to treat the independent parts as indivisible sections: so-called *critical sections*. This can restrict the number of choices due to parallelism, especially when the independent parts are large. To implement critical sections semaphores can be used.

Semaphores are protected variables to restrict access to shared resources in a multi-thread environment and are first introduced in [DYK65]. If only one is allowed access a binary semaphore is used. Binary semaphores are used to implement mutual exclusion algorithms. It can prevent critical code from being executed in parallel.

Two actions can be performed on semaphores: P and V actions. The name P action is derived from the Dutch fictional word blend *prolaag*, which means 'try-to-decrease'. If a P action is executed, it waits until the semaphore can be decreased. After the P action, it is allowed to execute some critical code. The name V action is derived from the Dutch word *verhoog*, which means 'increase'. If the critical code is executed, the semaphore is increased with a V action. This gives other critical code the opportunity to be executed afterward.

The binary semaphores make it possible to treat parts of certain processes as *critical sections*. This limits the number of choices due to parallelism. For example, if two parts are run in parallel ($a_1 \cdot a_2 \cdot ... \cdot a_{n-1} \cdot a_n \ || \ b_1 \cdot b_2 \cdot ... \cdot b_{m-1} \cdot b_m$), it generates $(n+1)(m+1)$ states. If the same two parts are treated as two critical sections, it only generates $2n + 2m$ states. Only one choice needs to be made, which critical section goes first. The original and reduced transition system are shown in figure B.1.

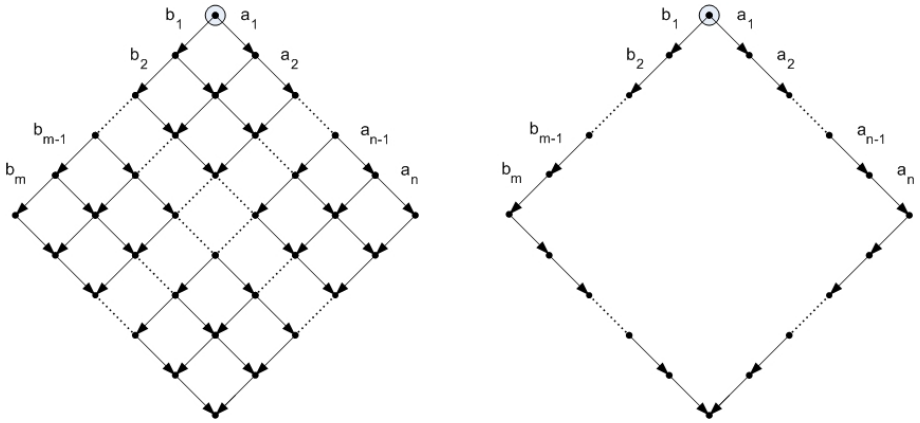

Figure B.1: Transition system with two parallel processes: original (left) and reduction using binary semaphores (right)

Binary semaphores can easily be implemented in process algebra. A process `Semaphore` needs to be build, which acts as a semaphore receiving P and V actions in turn. This process is run in parallel with the original specification. The P and V actions can then be implemented as communications with process `Semaphore`. The specification is shown below.

```
sort    SemaphoreAction
func    P,V:→SemaphoreAction

act     comm_Semaphore: SemaphoreAction
        send_Semaphore: SemaphoreAction
        recv_Semaphore: SemaphoreAction

comm    send_Semaphore | recv_Semaphore = comm_Semaphore
```

```
proc    Semaphore() =
        recv_Semaphore(P) · recv_Semaphore(V) ·
        Semaphore()
```

Of course, there are limitations to this technique, since it is not allowed to use binary semaphores randomly to reduce parallelism. Due to careless use of binary semaphores, it is possible to introduce deadlocks and it is possible to remove too many traces (resulting in a loss of system behaviour).

The first problem occurs if direct communication is possible between two processes with critical sections. A simple example is shown in figure B.2. Two processes (`p` and `q`) are run in parallel and both processes have a critical section. A deadlock occurs, because process `p` (or process `q`) will wait forever trying to communicate with the other process, which is waiting for a P action.

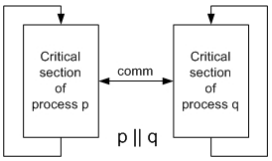

Figure B.2: Direct communication

The second problem occurs if two processes (or more) with critical sections can influence each other's behaviour. All process actions, which are responsible for this, need to be in different critical sections (or outside critical sections). Otherwise system behaviour is lost by the reduction. Note that it can be very tricky to conclude, how and where the behaviour of one process can be influenced by another process. It can be influenced via several other parallel processes.

In figure B.3 a simple example is given for the loss of system behaviour. Two processes (`p` and `q`) are run in parallel and both processes have critical sections. A third process `Memory` is used to hold a state variable. Process `p` sets the state variable once and process `q` reads the state variable twice during one loop. If only one critical section is used for process `q`, than it is impossible to set the state variable between two read actions. Some system behaviour is lost, since this behaviour is valid without the introduction of critical sections. This problem can be solved by putting both read actions in a different critical section.

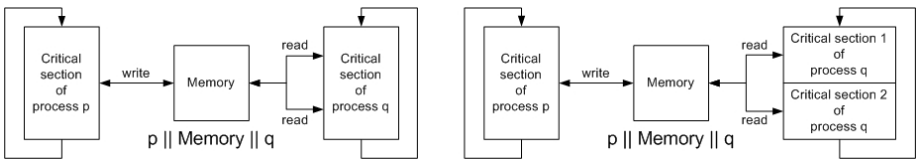

Figure B.3: Loss of behaviour: Problem (left) and solution (right)

B.3 Conclusion

Binary semaphores can be used to create critical sections in an attempt to reduce the generated transition system without losing system behaviour. These indivisible sections restrict the number of choices due to parallelism.

The validity of this technique can only be guaranteed, if a critical section is allowed to perform its actions independent of the other critical sections. The technique is valid under the following restrictions:

- *Processes with critical sections are not allowed to have direct communication.*
- *Each process action, which effects the behaviour of another critical section or is effected by it, needs to be in its own critical section.*

B.4 Steering robot

For the steering robot a binary semaphore is also used to restrict the number of choices due to parallelism. Critical sections are introduced in two parallel processes: `MACS(macs1)` and `MACS(macs2)`. These two processes communicate via canbus only and do not have direct communication[1].

The canbus is a delayed buffer. One proces writes to the buffer, while the other reads from that same buffer with a small delay. Both actions need to be in a different critical section to maintain all possible system behaviour. This means that the processes need to be split in two. The exact location of the division is unimportant, as long as the actions are seperated from each other. Even the size of the generated transition system is independent of the location of the division.

The processes of the steering robot are schematically shown in figure B.4.

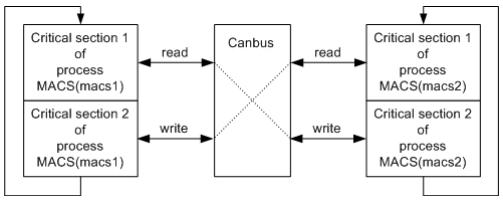

Figure B.4: Steering robot

Due to the use of binary semaphores the number of generated states is reduced by a factor 10. The original 5 million states are reduced to half a million states without losing system behaviour. With this smaller transition system, it is now possible to use the tools without the need of hiding too much information, making it a lot easier to find errors in the specification.

[1]Note that the model does not need an implementation of the dynamics of the steering robot to validate the requirements, otherwise there would be another indirect communication possible via the dynamics: reading sensor angle and performing actuator action. These actions would than lead to additional critical sections.

Appendix C

Process algebraic tools

The specification of the controller is built in μCRL. For the specification, analysis and verification several tools are used. In this appendix the used tools are briefly described.

C.1 μCRL toolset

Most tools used are in the μCRL toolset developed at CWI[1]. The following tools are part of this toolset:

- *Mcrl*: This tool checks if the μCRL specification is well-formed. It can also transform μCRL specifications to so-called linear process operators (LPOs). All other tools work with LPOs. The LPO is outputted in a TBF file.
- *Msim*: This tool can simulate the behavior of the system step by step and interactively. It uses an LPO or a μCRL specification. Unfortunately not all functionality can be used in a Windows environment.
- LPO Reduction tools: These tools try to reduce the complexity by rewriting the LPO and by eliminating specific μCRL elements. Several reduction tools are included: *Rewr*, *ConstElm*, *ParElm*, *SumElm* and *StructElm*.
- *Confcheck*: This tool tries to reduce the LPO by confluent τ reduction.
- *Instantiator*: This tool reads an LPO and generates a transition system.
- *LtsMin*: This tool can be used to reduce transition systems. One of the available reduction techniques is branching bisimulation.

The dependence between the different tools in the toolset is shown in figure C.1.

The toolset can be downloaded from http://homepages.cwi.nl/~mcrl/mutool.html. Unfortunately the toolset is developed on a Unix platform, making it difficult to run the toolset on a Windows platform. The installation procedures for Windows can be found at http://www.win.tue.nl/oas.

[1] Centrum voor Wiskunde en Informatica

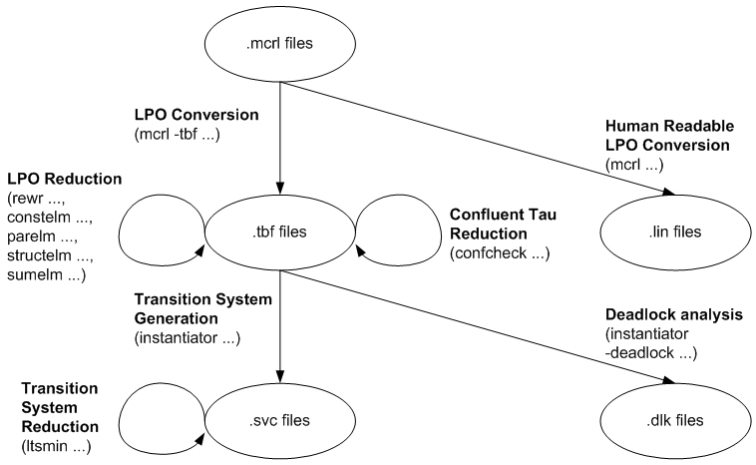

Figure C.1: μCRL toolset dependence

C.2 Cæsar Aldébaran Development Package

Some tools are used of the Cæsar Aldébaran Development Package (CADP). This toolset is developed at INRIA[2]. The tools can best be used from within the GUI *Eucalyptus*, but can also be used from command line. The following tools are mainly used:

- *Exp.Open*: Tool capable of generating a combined transition system from several communicating transition systems.
- *Evaluator*: Model-checker for regular alternation-free mu-calculus formulas on generated transition systems. It can also give counterexamples to disprove certain modal formulae.

The toolset can be downloaded from http://www.inrialpes.fr/vasy/cadp. This toolset can only be run on a Unix platform.

C.3 Visualization tools

For the visualization of the transition systems interactive tools are used, which are developed at TU/e. The following tools are used:

- *FSM Visualizer*: The FSM Visualizer can be used to visualize transition systems in 3D. The output is visualized on screen and can be explored interactively. The tool can be downloaded from http://www.win.tue.nl/∼fvham.
- *NoodleView*: NoodleView can be used to visualize the state variables of transition systems and to explore them interactively. The tool can be downloaded from http://www.win.tue.nl/∼apretori.

[2]Institut National de Recherche en Informatique et en Automatique

Appendix D

Formal methods code

In this appendix the complete specification of the GDA is given in μCRL. The verified modal formulae are also given.

D.1 Specification

```
%=====================================================
% Rules to reason about bools
%=====================================================

sort Bool
func T,F :-> Bool

map not:Bool->Bool
    or,and:Bool#Bool -> Bool
    eq:Bool#Bool->Bool

var x:Bool
rew not(T)=F
    not(F)=T
    and(T,x)=x
    and(x,T)=x
    and(x,F)=F
    and(F,x)=F
    or(T,x)=T
    or(x,T)=T
    or(x,F)=x
    or(F,x)=x
    eq(x,T)=x
    eq(T,x)=x
    eq(F,x)=not(x)
    eq(x,F)=not(x)

%=====================================================
% MacsIndex Type
%=====================================================

sort MacsIndex
```

```
func macs1,macs2 :-> MacsIndex

map other: MacsIndex -> MacsIndex
    eq: MacsIndex#MacsIndex->Bool

rew other(macs1)=macs2
    other(macs2)=macs1
    eq(macs1,macs1) = T
    eq(macs1,macs2) = F
    eq(macs2,macs1) = F
    eq(macs2,macs2) = T

%=====================================================
% UserAction type
%=====================================================

sort UserAction

func userStart, userStop, userEmergency, userReset :-> UserAction

map eq:UserAction#UserAction -> Bool

rew eq(userStart,userStart)         = T
    eq(userStart,userStop)          = F
    eq(userStart,userEmergency)     = F
    eq(userStart,userReset)         = F
    eq(userStop,userStart)          = F
    eq(userStop,userStop)           = T
    eq(userStop,userEmergency)      = F
    eq(userStop,userReset)          = F
    eq(userEmergency,userStart)     = F
    eq(userEmergency,userStop)      = F
    eq(userEmergency,userEmergency) = T
    eq(userEmergency,userReset)     = F
    eq(userReset,userStart)         = F
    eq(userReset,userStop)          = F
    eq(userReset,userEmergency)     = F
    eq(userReset,userReset)         = T

%=====================================================
% Semaphore type
%=====================================================

sort SemaphoreAction
func P,V :-> SemaphoreAction

map  eq:SemaphoreAction#SemaphoreAction -> Bool

rew  eq(P,P) = T
     eq(P,V) = F
     eq(V,P) = F
     eq(V,V) = T
```

```
%=====================================================
% Mode type
%=====================================================

sort Mode

func modeMenu, modeDrive, modeError: -> Mode

map eq:Mode#Mode -> Bool

rew eq(modeMenu,modeMenu)   = T
    eq(modeMenu,modeDrive)  = F
    eq(modeMenu,modeError)  = F
    eq(modeDrive,modeMenu)  = F
    eq(modeDrive,modeDrive) = T
    eq(modeDrive,modeError) = F
    eq(modeError,modeMenu)  = F
    eq(modeError,modeDrive) = F
    eq(modeError,modeError) = T

%=====================================================
% ModeOther type
%=====================================================

sort ModeOther

func otherMenu, otherMenuError, otherDrive, otherError: -> ModeOther

map eq:ModeOther#ModeOther -> Bool

rew eq(otherMenu,otherMenu)             = T
    eq(otherMenu,otherDrive)            = F
    eq(otherMenu,otherError)            = F
    eq(otherMenu,otherMenuError)        = F
    eq(otherDrive,otherMenu)            = F
    eq(otherDrive,otherDrive)           = T
    eq(otherDrive,otherError)           = F
    eq(otherDrive,otherMenuError)       = F
    eq(otherError,otherMenu)            = F
    eq(otherError,otherDrive)           = F
    eq(otherError,otherError)           = T
    eq(otherError,otherMenuError)       = F
    eq(otherMenuError,otherMenu)        = F
    eq(otherMenuError,otherDrive)       = F
    eq(otherMenuError,otherError)       = F
    eq(otherMenuError,otherMenuError)   = T

%=====================================================
% Error type
%=====================================================

sort Error

func errorNo, errorMACS, errorActuator, errorSensor,
```

```
                errorController: -> Error

map eq:Error#Error -> Bool

rew eq(errorNo,errorNo)                 = T
    eq(errorNo,errorMACS)               = F
    eq(errorNo,errorActuator)           = F
    eq(errorNo,errorSensor)             = F
    eq(errorNo,errorController)         = F
    eq(errorMACS,errorNo)               = F
    eq(errorMACS,errorMACS)             = T
    eq(errorMACS,errorActuator)         = F
    eq(errorMACS,errorSensor)           = F
    eq(errorMACS,errorController)       = F
    eq(errorActuator,errorNo)           = F
    eq(errorActuator,errorMACS)         = F
    eq(errorActuator,errorActuator)     = T
    eq(errorActuator,errorSensor)       = F
    eq(errorActuator,errorController)   = F
    eq(errorSensor,errorNo)             = F
    eq(errorSensor,errorMACS)           = F
    eq(errorSensor,errorActuator)       = F
    eq(errorSensor,errorSensor)         = T
    eq(errorSensor,errorController)     = F
    eq(errorController,errorNo)         = F
    eq(errorController,errorMACS)       = F
    eq(errorController,errorActuator)   = F
    eq(errorController,errorSensor)     = F
    eq(errorController,errorController) = T

%=====================================================
% SetpointValue type
%=====================================================

sort SetpointValue

func setpointValueL1, setpointValue0, setpointValueR1 :-> SetpointValue

map eq:SetpointValue#SetpointValue -> Bool

rew eq(setpointValueL1,setpointValueL1)     = T
    eq(setpointValueL1,setpointValue0)      = F
    eq(setpointValueL1,setpointValueR1)     = F
    eq(setpointValue0,setpointValueL1)      = F
    eq(setpointValue0,setpointValue0)       = T
    eq(setpointValue0,setpointValueR1)      = F
    eq(setpointValueR1,setpointValueL1)     = F
    eq(setpointValueR1,setpointValue0)      = F
    eq(setpointValueR1,setpointValueR1)     = T

%=====================================================
% SensorIndex type
%=====================================================
```

```
sort SensorIndex

func sensor1, sensor2, sensor3: -> SensorIndex

map eq:SensorIndex#SensorIndex -> Bool

rew eq(sensor1,sensor1) = T
    eq(sensor1,sensor2) = F
    eq(sensor1,sensor3) = F
    eq(sensor2,sensor1) = F
    eq(sensor2,sensor2) = T
    eq(sensor2,sensor3) = F
    eq(sensor3,sensor1) = F
    eq(sensor3,sensor2) = F
    eq(sensor3,sensor3) = T

%=====================================================
% SensorAngle type
%=====================================================

sort SensorAngle

func sensorAngleL1, sensorAngle0, sensorAngleR1 :-> SensorAngle

map eq:SensorAngle#SensorAngle -> Bool

rew eq(sensorAngleL1,sensorAngleL1) = T
    eq(sensorAngleL1,sensorAngle0)  = F
    eq(sensorAngleL1,sensorAngleR1) = F
    eq(sensorAngle0,sensorAngleL1)  = F
    eq(sensorAngle0,sensorAngle0)   = T
    eq(sensorAngle0,sensorAngleR1)  = F
    eq(sensorAngleR1,sensorAngleL1) = F
    eq(sensorAngleR1,sensorAngle0)  = F
    eq(sensorAngleR1,sensorAngleR1) = T

map match:SensorAngle#SensorAngle -> Bool

rew match(sensorAngleL1,sensorAngleL1)  = T
    match(sensorAngleL1,sensorAngle0)   = T
    match(sensorAngleL1,sensorAngleR1)  = F
    match(sensorAngle0,sensorAngleL1)   = T
    match(sensorAngle0,sensorAngle0)    = T
    match(sensorAngle0,sensorAngleR1)   = T
    match(sensorAngleR1,sensorAngleL1)  = F
    match(sensorAngleR1,sensorAngle0)   = T
    match(sensorAngleR1,sensorAngleR1)  = T

map min:SensorAngle#SensorAngle -> SensorAngle

rew min(sensorAngleL1,sensorAngleL1)    = sensorAngleL1
    min(sensorAngleL1,sensorAngle0)     = sensorAngleL1
    min(sensorAngleL1,sensorAngleR1)    = sensorAngleL1
    min(sensorAngle0,sensorAngleL1)     = sensorAngleL1
```

```
    min(sensorAngle0,sensorAngle0)      = sensorAngle0
    min(sensorAngle0,sensorAngleR1)     = sensorAngle0
    min(sensorAngleR1,sensorAngleL1)    = sensorAngleL1
    min(sensorAngleR1,sensorAngle0)     = sensorAngle0
    min(sensorAngleR1,sensorAngleR1)    = sensorAngleR1

map max:SensorAngle#SensorAngle -> SensorAngle

rew max(sensorAngleL1,sensorAngleL1)    = sensorAngleL1
    max(sensorAngleL1,sensorAngle0)     = sensorAngle0
    max(sensorAngleL1,sensorAngleR1)    = sensorAngleR1
    max(sensorAngle0,sensorAngleL1)     = sensorAngle0
    max(sensorAngle0,sensorAngle0)      = sensorAngle0
    max(sensorAngle0,sensorAngleR1)     = sensorAngleR1
    max(sensorAngleR1,sensorAngleL1)    = sensorAngleR1
    max(sensorAngleR1,sensorAngle0)     = sensorAngleR1
    max(sensorAngleR1,sensorAngleR1)    = sensorAngleR1

map median:SensorAngle#SensorAngle#SensorAngle -> SensorAngle

var sv1, sv2, sv3: SensorAngle
rew median(sv1,sv2,sv3) = max( max( min(sv1,sv2) , min(sv1,sv3) ) , min(sv2,sv3) )

map fault:SensorAngle#SensorAngle#SensorAngle#SensorAngle -> Bool

var svMedian, sv1, sv2, sv3: SensorAngle
rew fault(svMedian,sv1,sv2,sv3) = not( and( and( match(svMedian,sv1) ,
            match(svMedian,sv2) ) , match(svMedian,sv3) ) )

%=====================================================
% ActuatorAction type
%=====================================================

sort ActuatorAction

func actuatorActionL1, actuatorAction0, actuatorActionR1,
            actuatorActionError :-> ActuatorAction

map eq:ActuatorAction#ActuatorAction -> Bool

rew eq(actuatorActionL1,actuatorActionL1)       = T
    eq(actuatorActionL1,actuatorAction0)        = F
    eq(actuatorActionL1,actuatorActionR1)       = F
    eq(actuatorActionL1,actuatorActionError)    = F
    eq(actuatorAction0,actuatorActionL1)        = F
    eq(actuatorAction0,actuatorAction0)         = T
    eq(actuatorAction0,actuatorActionR1)        = F
    eq(actuatorAction0,actuatorActionError)     = F
    eq(actuatorActionR1,actuatorActionL1)       = F
    eq(actuatorActionR1,actuatorAction0)        = F
    eq(actuatorActionR1,actuatorActionR1)       = T
    eq(actuatorActionR1,actuatorActionError)    = F
    eq(actuatorActionError,actuatorActionL1)    = F
    eq(actuatorActionError,actuatorAction0)     = F
```

```
    eq(actuatorActionError,actuatorActionR1)    = F
    eq(actuatorActionError,actuatorActionError) = T

%.....................................................
% action
%
% This method computes the controller action from measured
% angle and setpoint respectively.
%.....................................................
map action:SensorAngle#SetpointValue -> ActuatorAction

rew action(sensorAngleL1,setpointValueL1)   = actuatorAction0
    action(sensorAngleL1,setpointValue0)    = actuatorActionR1
    action(sensorAngleL1,setpointValueR1)   = actuatorActionR1
    action(sensorAngle0,setpointValueL1)    = actuatorActionL1
    action(sensorAngle0,setpointValue0)     = actuatorAction0
    action(sensorAngle0,setpointValueR1)    = actuatorActionR1
    action(sensorAngleR1,setpointValueL1)   = actuatorActionL1
    action(sensorAngleR1,setpointValue0)    = actuatorActionL1
    action(sensorAngleR1,setpointValueR1)   = actuatorAction0

%=====================================================
% Semaphore actions
%=====================================================

act recv_Semaphore: SemaphoreAction
    send_Semaphore: SemaphoreAction

act comm_Semaphore: SemaphoreAction

comm send_Semaphore | recv_Semaphore = comm_Semaphore

%=====================================================
% Semaphore Process
%=====================================================

proc Semaphore =
    recv_Semaphore(P) . recv_Semaphore(V) . Semaphore

%=====================================================
% Time actions
%=====================================================

act comm_time
    send_time
    recv_time

comm send_time | recv_time = comm_time

%=====================================================
% Time processes
%=====================================================

%.....................................................
```

```
% Time
%
% MACSes are synchronous
%.....................................................
%proc Time(i: MacsIndex) =
%   (send_time
%       <| eq(i,macs1) |>
%    recv_time)

%.....................................................
% Time
%
% MACSes are asynchronous
%.....................................................
proc Time(i: MacsIndex) =
    (
        (
            send_time.send_time +
            send_time.send_time.send_time
        )
            <| eq(i,macs1) |>
        (
            recv_time.recv_time +
            recv_time.recv_time.recv_time
        )
    )


%=====================================================
% User actions
%=====================================================

act recv_Spv_UI: MacsIndex # UserAction

%=====================================================
% User processes
%=====================================================


%=====================================================
% Setpoint actions
%=====================================================

act recv_Ctrl_Setpoint: MacsIndex # SetpointValue

%=====================================================
% Setpoint processes
%=====================================================


%=====================================================
% System actions
%=====================================================
```

```
act recv_Angle_Sensor: MacsIndex # SensorIndex # SensorAngle
    recv_Spv_ActuatorState: MacsIndex # Bool
    send_Spv_ActuatorEnable: MacsIndex # Bool
    send_Ctrl_ActuatorAction: MacsIndex # ActuatorAction

%=====================================================
% System processes
%=====================================================



%=====================================================
% Internal Buffer actions
%=====================================================

act recv_Spv_Mode: MacsIndex # Mode
    recv_Mon_Mode: MacsIndex # Mode
    send_Spv_Mode: MacsIndex # Mode
    recv_Spv_Error: MacsIndex # Error
    recv_Mon_Error: MacsIndex # Error
    send_Spv_Error: MacsIndex # Error
    send_Spv_Reset: MacsIndex
    recv_Spv_SensorError: MacsIndex # Bool
    send_Angle_SensorError: MacsIndex # Bool
    recv_Ctrl_SensorValue: MacsIndex # SensorAngle
    send_Angle_SensorValue: MacsIndex # SensorAngle

act comm_Mode_Get: MacsIndex # Mode
    comm_Mode_Get_Mon: MacsIndex # Mode
    comm_Mode_Set: MacsIndex # Mode
    comm_Error_Get: MacsIndex # Error
    comm_Error_Get_Mon: MacsIndex # Error
    comm_Error_Set: MacsIndex # Error
    comm_Error_Reset: MacsIndex
    comm_SensorError_Get: MacsIndex # Bool
    comm_SensorError_Set: MacsIndex # Bool
    comm_SensorValue_Get: MacsIndex # SensorAngle
    comm_SensorValue_Set: MacsIndex # SensorAngle

act recv_Buf_Mode: MacsIndex # Mode
    send_Buf1_Mode: MacsIndex # Mode
    send_Buf2_Mode: MacsIndex # Mode
    recv_Buf_Error: MacsIndex # Error
    send_Buf1_Error: MacsIndex # Error
    send_Buf2_Error: MacsIndex # Error
    recv_Buf_Reset: MacsIndex
    recv_Buf_SensorError: MacsIndex # Bool
    send_Buf_SensorError: MacsIndex # Bool
    recv_Buf_SensorValue: MacsIndex # SensorAngle
    send_Buf_SensorValue: MacsIndex # SensorAngle

comm send_Buf1_Mode | recv_Spv_Mode = comm_Mode_Get
     send_Buf2_Mode | recv_Mon_Mode = comm_Mode_Get_Mon
     recv_Buf_Mode | send_Spv_Mode = comm_Mode_Set
```

```
     send_Buf1_Error | recv_Spv_Error = comm_Error_Get
     send_Buf2_Error | recv_Mon_Error = comm_Error_Get_Mon
     recv_Buf_Error | send_Spv_Error = comm_Error_Set
     recv_Buf_Reset | send_Spv_Reset = comm_Error_Reset
     send_Buf_SensorError | recv_Spv_SensorError = comm_SensorError_Get
     recv_Buf_SensorError | send_Angle_SensorError = comm_SensorError_Set
     send_Buf_SensorValue | recv_Ctrl_SensorValue = comm_SensorValue_Get
     recv_Buf_SensorValue | send_Angle_SensorValue = comm_SensorValue_Set


%=====================================================
% Internal Buffer Processes
%
% Internal buffer processes are used to store certain
% values: sensorerror, sensorvalue, mode, error and alive
%=====================================================

proc Buffers(i: MacsIndex) =
    Buffer_Mode(i, modeMenu) || Buffer_Error(i, errorNo) ||
    Buffer_SensorError(i, F) || Buffer_SensorValue(i, sensorAngle0)

proc Buffer_Mode(i: MacsIndex, m: Mode) =
    sum(n:Mode, recv_Buf_Mode(i,n) . Buffer_Mode(i,n)) +
    send_Buf1_Mode(i,m) . Buffer_Mode(i,m) +
    send_Buf2_Mode(i,m) . Buffer_Mode(i,m)

proc Buffer_Error(i: MacsIndex, e: Error) =
    sum(f:Error, recv_Buf_Error(i,f) . Buffer_Error(i,f)) +
    send_Buf1_Error(i,e) . Buffer_Error(i,e) +
    send_Buf2_Error(i,e) . Buffer_Error(i,e) +
    recv_Buf_Reset(i) . Buffer_Error(i,errorNo)

proc Buffer_SensorError(i: MacsIndex, f: Bool) =
    sum(g:Bool, recv_Buf_SensorError(i,g) . Buffer_SensorError(i,g)) +
    send_Buf_SensorError(i,f) . Buffer_SensorError(i,f)

proc Buffer_SensorValue(i: MacsIndex, sv: SensorAngle) =
    sum(sv_new:SensorAngle, recv_Buf_SensorValue(i,sv_new) . Buffer_SensorValue(i,sv_new)) +
    send_Buf_SensorValue(i,sv) . Buffer_SensorValue(i,sv)

%=====================================================
% Canbus Actions
%=====================================================

act send_Can_Mode: MacsIndex # ModeOther
    recv_Can_Mode: MacsIndex # ModeOther
    send_Can_Alive: MacsIndex # Bool
    recv_Can_Alive: MacsIndex # Bool

act recv_Mon_Mode_Other: MacsIndex # ModeOther
    send_Mon_Mode_Other: MacsIndex # ModeOther
    recv_Mon_Alive: MacsIndex # Bool
    send_Mon_Alive: MacsIndex # Bool
```

```
act comm_Mode_Other_Get: MacsIndex # ModeOther
    comm_Mode_Other_Set: MacsIndex # ModeOther
    comm_Alive_Get: MacsIndex # Bool
    comm_Alive_Set: MacsIndex # Bool

comm send_Can_Mode | recv_Mon_Mode_Other = comm_Mode_Other_Get
     recv_Can_Mode | send_Mon_Mode_Other = comm_Mode_Other_Set
     send_Can_Alive | recv_Mon_Alive = comm_Alive_Get
     recv_Can_Alive | send_Mon_Alive = comm_Alive_Set

%=====================================================
% Canbus Processes
%=====================================================

proc Can_Mode(i: MacsIndex, m: ModeOther, m_prev: ModeOther) =
    sum(n:ModeOther, recv_Can_Mode(i,n) . Can_Mode(i,n,m)) +
    send_Can_Mode(i,m_prev) . Can_Mode(i,m,m_prev)

proc Can_Alive(i: MacsIndex, e: Bool, e_prev: Bool) =
    sum(f:Bool, recv_Can_Alive(i,f) . Can_Alive(i,f,e)) +
    send_Can_Alive(i,e_prev) . Can_Alive(i,e,e_prev)

proc Canbus =
    Can_Alive(macs1,T,T) ||
    Can_Alive(macs2,T,T) ||
    Can_Mode(macs1,otherMenu,otherMenu) ||
    Can_Mode(macs2,otherMenu,otherMenu)

%=====================================================
% Monitor actions
%=====================================================

%=====================================================
% Monitor processes
%=====================================================

proc Monitor_In(i:MacsIndex) =
    Monitor_In_Alive(i) . Monitor_In_Mode(i)

proc Monitor_In_Alive(i: MacsIndex) =
    sum(f: Bool, recv_Mon_Alive(other(i),f) .
        (tau
            <| f |>
        send_Spv_Error(i,errorMACS))
    )

proc Monitor_In_Mode(i: MacsIndex) =
    sum(m: ModeOther, recv_Mon_Mode_Other(other(i),m) .
        (
            (tau                          <| eq(m,otherMenu)      |> delta) +
            (send_Spv_Error(i,errorMACS)  <| eq(m,otherMenuError) |> delta) +
            (tau                          <| eq(m,otherDrive)     |> delta) +
            (send_Spv_Error(i,errorMACS) . send_Spv_Mode(i,modeError)
                <| eq(m,otherError) |> delta)
```

```
        )
    )

proc Monitor_Out(i: MacsIndex) =
    send_Mon_Alive(i,T) . Monitor_Out_Mode(i)


proc Monitor_Out_Mode(i: MacsIndex) =
    sum(e: Error, recv_Mon_Error(i,e) .
        sum(m: Mode, recv_Mon_Mode(i,m) .
            (
                (
                    (send_Mon_Mode_Other(i,otherMenu)
                        <| eq(e,errorNo) |>
                    send_Mon_Mode_Other(i,otherMenuError))
                    <| eq(m,modeMenu)  |> delta
                ) +
                (send_Mon_Mode_Other(i,otherDrive)
                    <| eq(m,modeDrive) |> delta
                ) +
                (send_Mon_Mode_Other(i,otherError)
                    <| eq(m,modeError) |> delta
                )
            )
        )
    )


%=====================================================
% Angle process
%=====================================================

proc Angle(i: MacsIndex) =
    sum(sv1:SensorAngle, recv_Angle_Sensor(i,sensor1,sv1) .
        sum(sv2:SensorAngle, recv_Angle_Sensor(i,sensor2,sv2) .
            sum(sv3:SensorAngle, recv_Angle_Sensor(i,sensor3,sv3) .
                send_Angle_SensorValue(i,median(sv1,sv2,sv3)) .
                send_Angle_SensorError(i,fault(median(sv1,sv2,sv3),sv1,sv2,sv3))
            )
        )
    )

%=====================================================
% Supervisor actions
%=====================================================

act errorMessage: MacsIndex # Error

%=====================================================
% Supervisor processes
%
% Supervisor takes care of:
% -System check
% -Controller mode selection
```

```
%=====================================================

proc Supervisor(i: MacsIndex) =
    Supervisor_Systemcheck(i) .
    Supervisor_Mode(i)

%-----------------------------------------------------
% Supervisor system check
%-----------------------------------------------------
proc Supervisor_Systemcheck(i: MacsIndex) =
    Supervisor_Sensor(i) .
    Supervisor_Actuator(i)

%.....................................................
% Check if the sensor gave an error (internal sensor error buffer)
%.....................................................
proc Supervisor_Sensor(i: MacsIndex) =
    sum(f: Bool, recv_Spv_SensorError(i,f) .
        (send_Spv_Error(i,errorSensor)
        <| f |>
        tau)
    )

%.....................................................
% Check the actuator's state (hardware actuator buffer) and
% turn it off if it's not functioning
%.....................................................
proc Supervisor_Actuator(i: MacsIndex) =
    sum(f: Bool, recv_Spv_ActuatorState(i,f) .
        (send_Spv_Error(i,errorActuator) . send_Spv_ActuatorEnable(i,F)
        <| not(f) |>
         send_Spv_ActuatorEnable(i,T) )
    )

%-----------------------------------------------------
% Controller mode selection
%-----------------------------------------------------
proc Supervisor_Mode(i: MacsIndex) =
    sum(m: Mode, recv_Spv_Mode(i,m) .
        (
            (Supervisor_Menu(i)  <| eq(m,modeMenu)  |> delta) +
            (Supervisor_Drive(i) <| eq(m,modeDrive) |> delta) +
            (Supervisor_Error(i) <| eq(m,modeError) |> delta)
        )
    )

%.....................................................
% Menu mode:
% - start: Driving mode (if no errors in system)
% - stop: Menu mode
% - emergency: Error mode
% - reset: Menu mode, reset internal error buffer
%.....................................................
proc Supervisor_Menu(i: MacsIndex) =
```

```
    sum(e: Error, recv_Spv_Error(i,e) .
        sum(ua: UserAction, recv_Spv_UI(i,ua) .
            (
                (Supervisor_Start(i,e)       <| eq(ua,userStart)     |> delta) +
                (tau                         <| eq(ua,userStop)      |> delta) +
                (send_Spv_Mode(i,modeError) . send_Spv_Error(i,errorNo)
                    <| eq(ua,userEmergency) |> delta) +
                (send_Spv_Reset(i)           <| eq(ua,userReset)     |> delta)
            )
        )
    )

proc Supervisor_Start(i: MacsIndex, e: Error) =
    (send_Spv_Mode(i,modeDrive)
        <|eq(e,errorNo)|>
    errorMessage(i,e))

%.................................................
% Drive mode:
% - systemcheck error: Error mode
% - start/reset: Driving mode
% - stop: Menu mode
% - emergency: Error mode
%.................................................
proc Supervisor_Drive(i: MacsIndex) =
    sum(e: Error, recv_Spv_Error(i,e) .
        (sum(ua: UserAction, recv_Spv_UI(i,ua) .
            (
                (tau                         <| eq(ua,userStart)     |> delta) +
                (send_Spv_Mode(i,modeMenu)   <| eq(ua,userStop)      |> delta) +
                (send_Spv_Mode(i,modeError)  <| eq(ua,userEmergency) |> delta) +
                (send_Spv_Mode(i,modeMenu)   <| eq(ua,userReset)     |> delta)
            )
             )
        <| eq(e,errorNo) |>
         send_Spv_Mode(i,modeError) . Supervisor_Error(i)
        )
    )


%.................................................
% Error mode:
% - start/stop/emergency: Error mode
% - reset: Menu mode, reset internal error buffer
%.................................................
proc Supervisor_Error(i: MacsIndex) =
    sum(e: Error, recv_Spv_Error(i,e) .
        sum(ua: UserAction, recv_Spv_UI(i,ua) .
            (send_Spv_Reset(i) . send_Spv_Mode(i,modeMenu)
            <| eq(ua,userReset) |>
            errorMessage(i,e))
        )
    )
```

```
%=====================================================
% Controller actions
%=====================================================

act menu: MacsIndex
    drive: MacsIndex
    error: MacsIndex

%=====================================================
% Controller processes
%
% The controller takes care of the specific controller
% action to be taken
%=====================================================

proc Controller(i: MacsIndex) =
    sum(m: Mode, recv_Spv_Mode(i,m) .
        (
            ( menu(i)                      <| eq(m,modeMenu)  |> delta) +
            ( drive(i).Controller_Drive(i) <| eq(m,modeDrive) |> delta) +
            ( error(i).Controller_Error(i) <| eq(m,modeError) |> delta)
        )
    )

%-----------------------------------------------------
% Control actions in driving method
%-----------------------------------------------------
proc Controller_Drive(i: MacsIndex) =
    sum(sv: SensorAngle, recv_Ctrl_SensorValue(i,sv) .
        sum(s: SetpointValue, recv_Ctrl_Setpoint(i,s) .
            ( send_Ctrl_ActuatorAction(i,action(sv,s))+
              send_Spv_Error(i,errorController) )
        )
    )

%-----------------------------------------------------
% Control actions in error method
%-----------------------------------------------------
proc Controller_Error(i: MacsIndex) =
    sum(e: Error, recv_Spv_Error(i,e) .
        (send_Ctrl_ActuatorAction(i,actuatorAction0)
            <|eq(e,errorActuator)|>
        ( send_Ctrl_ActuatorAction(i,actuatorActionError)) +
          send_Spv_Error(i,errorController) )
    )

%=====================================================
% Main processes
%=====================================================

proc MACS(i: MacsIndex) =
    send_Semaphore(P) .
    Monitor_In(i) .
    Angle(i) . Supervisor(i) . Controller(i) .
```

```
    send_Semaphore(V) .
    send_Semaphore(P) .
    Monitor_Out(i) .
    send_Semaphore(V) .
    Time(i) . MACS(i)


proc System =
    MACS(macs1) ||
    Buffers(macs1) ||
    MACS(macs2) ||
    Buffers(macs2) ||
    Canbus ||
    Semaphore


%=====================================================
% Initialisation
%=====================================================

init

encap(
{

    %Timing mechanism: Comment if single MACS analysis is done
    send_time, recv_time,

    %Semaphore
    send_Semaphore, recv_Semaphore,

    %Canbus connection between both MACS
    send_Can_Mode, recv_Mon_Mode_Other, recv_Can_Mode, send_Mon_Mode_Other,
    send_Can_Alive, recv_Mon_Alive, recv_Can_Alive, send_Mon_Alive,

    %Internal buffers of a single MACS
    send_Buf1_Mode, recv_Spv_Mode,
    send_Buf2_Mode, recv_Mon_Mode,
    recv_Buf_Mode, send_Spv_Mode,
    send_Buf1_Error, recv_Spv_Error,
    send_Buf2_Error, recv_Mon_Error,
    recv_Buf_Error, send_Spv_Error,
    recv_Buf_Reset, send_Spv_Reset,
    send_Buf_SensorError, recv_Spv_SensorError,
    recv_Buf_SensorError, send_Angle_SensorError,
    send_Buf_SensorValue, recv_Ctrl_SensorValue,
    recv_Buf_SensorValue, send_Angle_SensorValue

},

        System

)
```

D.2 Modal formulae

Three requirements were verified with additional conditions to express that the second control system also behaves in the same way as the first control system after a user action. The verified modal formulae are given in the following subsections.

D.2.1 Requirement 2

$$
\begin{array}{l}
[\ \beta_{menu} \cdot \alpha^*_{other} \cdot \texttt{start}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{drive\}}(i)\ ]\ \mathrm{F} \\
\wedge \\
[\ \beta_{menu} \cdot \alpha^*_{other} \cdot \texttt{start}(i)\ ] \\
\langle\ \alpha^*_{other} \cdot (\ \texttt{drive}(i) \mid (\alpha_{errorknown} \cdot \alpha_{other} \cdot \texttt{menu}(i)))\ \rangle\ \mathrm{T} \\
\wedge \\
[\ \beta_{menu} \cdot \\
(\alpha^*_{other} \cdot \texttt{start}(i) \cdot \alpha^*_{other} \cdot \texttt{drive}(i))^+ \cdot \\
\alpha^*_{other} \cdot \texttt{start}(other(i)) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{drive\}}(other(i))\ ]\ \mathrm{F} \\
\wedge \\
[\ \beta_{menu} \cdot \\
(\alpha^*_{other} \cdot \texttt{start}(i) \cdot \alpha^*_{other} \cdot \texttt{drive}(i))^+ \cdot \\
\alpha^*_{other} \cdot \texttt{start}(other(i))\ ] \\
\langle\ \alpha^*_{other} \cdot (\ \texttt{drive}(other(i)) \mid (\alpha_{errorknown} \cdot \alpha_{other} \cdot \texttt{menu}(other(i))))\ \rangle\ \mathrm{T}
\end{array}
$$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode} \wedge \neg\alpha_{errorknown}$

D.2.2 Requirement 3

$$
\begin{array}{l}
[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \texttt{stop}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{menu\}}(i)\ ]\ \mathrm{F} \\
\wedge \\
[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \texttt{stop}(i)\ ]\langle\ \alpha^*_{other} \cdot \texttt{menu}(i)\ \rangle\ \mathrm{T} \\
\wedge \\
[\ \beta_{drive} \cdot \\
(\alpha^*_{other} \cdot \texttt{stop}(i) \cdot \alpha^*_{other} \cdot \texttt{menu}(i))^+ \cdot \\
\alpha^*_{other} \cdot \texttt{stop}(other(i)) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{menu\}}(other(i))\ ]\ \mathrm{F} \\
\wedge \\
[\ \beta_{drive} \cdot \\
(\alpha^*_{other} \cdot \texttt{stop}(i) \cdot \alpha^*_{other} \cdot \texttt{menu}(i))^+ \cdot \\
\alpha^*_{other} \cdot \texttt{stop}(other(i))\ ] \\
\langle\ \alpha^*_{other} \cdot \texttt{menu}(other(i))\ \rangle\ \mathrm{T}
\end{array}
$$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user} \wedge \neg\alpha_{error} \wedge \neg\alpha_{mode}$

D.2.3 Requirement 4

$$
\begin{array}{l}
[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \texttt{emergency}(i) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{error\}}(i)\ ]\ \mathrm{F} \\
\wedge \\
[\ \beta_{drive} \cdot \alpha^*_{other} \cdot \texttt{emergency}(i)\ ] \langle\ \alpha^*_{other} \cdot \texttt{error}(i)\ \rangle\ \mathrm{T} \\
\wedge \\
[\ \beta_{drive} \cdot \\
(\alpha^*_{other} \cdot \texttt{emergency}(i) \cdot \alpha^*_{other} \cdot \texttt{error}(i))^+ \cdot \\
\alpha^*_{other} \cdot \texttt{emergency}(other(i)) \cdot \alpha^*_{other} \cdot \alpha_{mode \setminus \{error\}}(other(i))\ ]\ \mathrm{F} \\
\wedge \\
[\ \beta_{drive} \cdot \\
(\alpha^*_{other} \cdot \texttt{emergency}(i) \cdot \alpha^*_{other} \cdot \texttt{error}(i))^+ \cdot \\
\alpha^*_{other} \cdot \texttt{emergency}(other(i))\ ] \\
\langle\ \alpha^*_{other} \cdot \texttt{error}(other(i))\ \rangle\ \mathrm{T}
\end{array}
$$

with $\alpha_{other} \equiv \top \wedge \neg\alpha_{user} \wedge \neg\alpha_{mode}$

Appendix E

Paper SCSC

In this appendix a paper is added, which is accepted for the 2007 Summer Computer Simulation Conference (SCSC) in San Diego, CA.