Safety-critical design of the Generic Driving Actuator: a hybrid approach

Citation for published version (APA):

Document status and date:
Published: 01/01/2007

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain.
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Safety-critical design of the Generic Driving Actuator

- A hybrid approach

L.L.F. Merkx\textsuperscript{1}, H.M. Duringhof\textsuperscript{2} and P.J.L. Cuijpers\textsuperscript{1}

\textsuperscript{1}Technische Universiteit Eindhoven, P.O.Box 513, 5600 MB Eindhoven, Netherlands
\textsuperscript{2}TNO Automotive, P.O.Box 756, 5700 AT Helmond, Netherlands
{L.L.F.Merkx,P.J.L.Cuijpers}@tue.nl

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 report, the design of the GDA is discussed and the interaction between its software and hardware components is analyzed from a safety point of view.

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 $\mu$CRL. 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 design a strategy 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.
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. The project background, the project description and the structure of the report 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 (sine-soid 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 research 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 report the design of the GDA is described. The objective of the research 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 report will be:

*The safety-critical design of the Generic Driving Actuator*

1.3 Structure of report

In this report, first the Generic Driving Actuator is introduced in chapter 2. The importance of safety is stressed.

Chapter 3 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 4 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. The formal methods are used to specify, design and verify the GDA’s software.

In chapter 5 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
1.3. Structure of report

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 6 contains the conclusions and further recommendations.

A short discussion is given in chapter 7 on the differences between computer science and mechanical engineering and how these fields should be combined. As this research also will show, combining both fields will improve the design proces and remove problems in an early phase by verifying requirements.
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.

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. 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.

1Modern 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.
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 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.

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 lets a vehicle follow a predefined trajectory. To test this system the steering, braking 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 threat to the test driver and to the environment. This potential threat 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 in 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 3.

- **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 4.

- **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 5.

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 Specification

The specification of the two robots are based on the needs of the driving tests and it needs to be comparable to the competition. The GDA needs to be a compact, light-weight robot, which is easy to install and operate. The exact specification is omitted in this version.
Chapter 3

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 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.

3.1 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 3.1). 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.

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 4: 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 5 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 3.2.

The reliability of hardware components during its lifetime changes. The failure rate characteristic, also known as the 'bathtub' curve [STO99], is shown in figure 3.2.

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 (\( \lambda \)). This failure rate will be used to characterize the reliability of a hardware component.

Figure 3.1: Reliability strategies during system lifetime [AL90]
3.2 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**.

![Static Redundancy (TMR)](image)

**Figure 3.3: 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 3.3 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 3.4. 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.
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 3.5.

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 3.6 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.
3.2. Fault-tolerant strategies

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 cannot 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 cannot be handled. The safety of the system can not be guaranteed as long as the failing component is not replaced by a new one.

![Figure 3.6: Hybrid redundancy (TMR with single spare)](image-url)
3.3 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 5. 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.

3.3.1 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 3.7.

![Figure 3.7: Fault-tolerant sensor design](image)

3.3.2 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 3.8.

![Fault-tolerant actuator design](image)

**Figure 3.8: 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.

### 3.3.3 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.

![Fault-tolerant control system design (Single spare)](image)

**Figure 3.9: Fault-tolerant control system design (Single spare)**

In figure 3.9 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 3.10. The fault detection is done by the other control system and the switch is distributed over the control systems.

![Figure 3.10: Fault-tolerant control system design (Dual control)](image)

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, then 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 4

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 converted to Simulink.

4.1 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 3.2 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 [AV185]**: Different code is used to perform the same tasks. This strategy is similar to the static redundancy strategy in section 3.2. 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 3.2. Recovery blocks are not commonly used, because the acceptance test is hard to implement.

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 3.1). During this research *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 5.
4.2 Interface description

In this section the interface description of the control system is given. The architecture of the GDA is schematically shown in figure 4.1. Although the GDA uses three robots, only one robot will be used throughout this chapter. This will not affect 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 4.1: Architecture of the GDA using redundant hardware](image)

4.2.1 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 5.

### 4.2.2 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 ($\tau$). 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 5.

Summarizing this behaviour, results in the transition diagram shown in figure 4.2.

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 \textit{LoggingPosition} and \textit{LoggingForce}. These logging signals will not be modeled and analyzed.
4.2. Interface description

4.2.3 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.
  - *Actuator Control signal*: This signal contains the control values, which are sent to the actuator.
  - *Actuator Enable signal*: The control system can enable the actuator, whenever it wants.
  - *Actuator State signal*: The control system receives information, about the actuator’s current state of operation.

- **Sensor signals**: The control system receives several sensor values:
  - *Sensor Position signals*: Three signals are provided, containing three independent position measurements.
  - *Sensor Force 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.

4.2.4 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.
4.3 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.

4.4 Specification

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

4.4.1 Introduction to \( \mu \)CRL

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 report the specification language \( \mu \)CRL is used, it is an extension of the process algebra ACP [BW90]. The main difference between \( \mu \)CRL and the classical process algebras is the formal treatment of data in \( \mu \)CRL. This makes \( \mu \)CRL more expressive than the classical process algebras.

A specification built in \( \mu \)CRL consists of two parts: data types and processes. The formal description of the syntax and semantics of \( \mu \)CRL 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 \( \mu \)CRL.

As an example the data type `Bool` and the equality function on `Bool` is implemented in \( \mu \)CRL. The data type `Bool` needs to be specified for every \( \mu \)CRL specification.

\[
\begin{align*}
\text{sort} & \quad \text{Bool} \\
\text{func} & \quad T,F: \rightarrow \text{Bool} \\
\text{map} & \quad \text{eq: } \text{Bool} \times \text{Bool} \rightarrow \text{Bool} \\
\text{var} & \quad b: \text{Bool} \\
\text{rew} & \quad \text{eq}(T,T) = T \\
& \quad \text{eq}(T,F) = F \\
& \quad \text{eq}(F,T) = F \\
& \quad \text{eq}(F,F) = T
\end{align*}
\]

Processes can be built from actions, operators and processes. The keyword `proc` is used to declare processes in \( \mu \)CRL.

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 \( \mu \)CRL: \( \delta \) and \( \tau \). The \( \delta \) is used for an inaction to indicate deadlock, i.e. the process stops performing actions. The \( \tau \) 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 \( \mu \)CRL: `sum` and \( (\triangleleft \triangleright) \). The first
operator is introduced to represent a (possibly infinite) choice between processes of a specific data type. For example \( \text{sum}(i : \mathbb{N}, p(i)) \) means that one specific process \( p \) can be executed with a random natural number as a parameter. The second introduced operator indicates conditional choice. \( p \triangleleft b \triangleright q \) 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 \texttt{comm}. Actions may only synchronize if the data parameters are equal. To force synchronization between two actions the keyword \texttt{encap} is used. Otherwise the actions can also execute on their own.

Another important operator is \texttt{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 \( \tau \) 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 \( \mu \text{CRL} \). The memory can receive both \( T \) or \( F \) and can send the boolean value, which is present in the memory.

\[
\begin{align*}
\text{act} & \quad \text{recv\_Bool: Bool} \\
& \quad \text{send\_Bool: Bool} \\
\text{proc} & \quad \text{Memory}(f: \text{Bool}) = \\
& \quad \text{sum}(f\_\text{new}: \text{Bool}, \text{recv\_Bool}(f\_\text{new}) \cdot \text{Memory}(f\_\text{new})) + \\
& \quad \text{send\_Bool}(f) \cdot \text{Memory}(f)
\end{align*}
\]

### 4.4.2 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: \( \text{MacIndex} \) and \( \text{SensorIndex} \). An actuator index is not needed for the two actuators, because each control system has its own actuator.

\[
\begin{align*}
\text{sort} & \quad \text{MacIndex} \\
\text{func} & \quad \text{macs1,macs2:} \rightarrow \text{MacIndex} \\
\text{map} & \quad \text{other: MacIndex} \rightarrow \text{MacIndex} \\
\text{rew} & \quad \text{other(macs1)} = \text{macs2} \\
& \quad \text{other(macs2)} = \text{macs1}
\end{align*}
\]

\[
\begin{align*}
\text{sort} & \quad \text{SensorIndex} \\
\text{func} & \quad \text{sensor1,sensor2,sensor3:} \rightarrow \text{SensorIndex}
\end{align*}
\]

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

\[
\text{sort} \quad \text{SensorValue} \\
\text{func} \quad \text{SensorValueL}, \text{SensorValue0}, \text{SensorValueR}: \to \text{SensorValue}
\]

\[
\text{map} \quad \text{match}: \text{SensorValue} \times \text{SensorValue} \to \text{Bool} \\
\text{map} \quad \text{min}: \text{SensorValue} \times \text{SensorValue} \to \text{SensorValue} \\
\text{map} \quad \text{max}: \text{SensorValue} \times \text{SensorValue} \to \text{SensorValue}
\]

\[
\text{map} \quad \text{median}: \text{SensorValue} \times \text{SensorValue} \times \text{SensorValue} \to \text{SensorValue} \\
\text{var} \quad \text{sv1}, \text{sv2}, \text{sv3}: \text{SensorValue} \\
\text{rew} \quad \text{median}(\text{sv1}, \text{sv2}, \text{sv3}) = \max(\max(\min(\text{sv1}, \text{sv2}), \min(\text{sv1}, \text{sv3})), \min(\text{sv2}, \text{sv3}))
\]

\[
\text{map} \quad \text{fault}: \text{SensorValue} \times \text{SensorValue} \times \text{SensorValue} \times \text{SensorValue} \to \text{SensorValue} \\
\text{var} \quad \text{svMedian}, \text{sv1}, \text{sv2}, \text{sv3}: \text{SensorValue} \\
\text{rew} \quad \text{fault}(\text{svMedian}, \text{sv1}, \text{sv2}, \text{sv3}) = \ldots
\]

\[
\text{sort} \quad \text{ActuatorAction} \\
\text{func} \quad \text{actuatorActionL}, \text{actuatorAction}, \text{actuatorActionR} \\
\text{actuatorAction0}, \text{actuatorActionR}: \to \text{ActuatorAction}
\]

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.

\[
\text{sort} \quad \text{UserAction} \\
\text{func} \quad \text{userStart}, \text{userStop}, \text{userEmergency}, \text{userReset}: \to \text{UserAction}
\]

\[
\text{sort} \quad \text{SetpointValue} \\
\text{func} \quad \text{setpointValueL}, \text{setpointValue0}, \text{setpointValueR}: \to \text{SetpointValue}
\]

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
   ...

4.4.3 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 C.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 com-
putation 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
4.4. Specification

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`.

```plaintext
proc Position(i:MacIndex) =
    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.

```plaintext
proc Supervisor(i:MacIndex) =
    Supervisor_Systemcheck(i) ·
    Supervisor_Mode(i)
```

The `Supervisor_Mode` process determines the operational mode from the pre-
vious 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.
Safety-critical software design

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 4.6.

```plaintext
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.

\[
\text{act} \quad \text{comm\_time} \; \text{send\_time} \; \text{recv\_time} \\
\text{comm} \quad \text{send\_time} \mid \text{recv\_time} = \text{comm\_time}
\]

\[
\text{proc} \quad \text{Time}(i:\text{MacsIndex}) = \\
\begin{pmatrix}
( \\
\text{send\_time} \cdot \text{send\_time} + \\
\text{send\_time} \cdot \text{send\_time} \cdot \text{send\_time}
) \\
\lhd \text{eq}(i,\text{macs1}) \rhd \\
( \\
\text{recv\_time} \cdot \text{recv\_time} + \\
\text{recv\_time} \cdot \text{recv\_time} \cdot \text{recv\_time}
)
\end{pmatrix}
\]

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 A. The process `MACS` is adjusted as follows:

\[
\text{proc} \quad \text{MACS}(i:\text{MacsIndex}) = \\
\text{send\_Semaphore(P)} \cdot \\
\text{Monitor\_In}(i) \cdot \text{Position}(i) \cdot \\
\text{Supervisor}(i) \cdot \text{Controller}(i) \cdot \\
\text{send\_Semaphore(V)} \cdot \text{send\_Semaphore(P)} \cdot \\
\text{Monitor\_Out}(i) \cdot \\
\text{send\_Semaphore(V)} \cdot \\
\text{Time}(i) \cdot \text{MACS}(i)
\]

All processes and their communication are shown in figure 4.3. In this figure only one control system is shown, since their behaviour is completely identical.
4.5 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 4.6 the final version of the specification is verified.

4.5.1 Problem 1

The first problem is related to the requirements. The last two requirements cannot 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 4.4. 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:
Communicate actuator actions over canbus: This method results in a delay for all actuator actions. This is unacceptable, 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 4.5. This strategy will certainly have a negative effect on controller performance, especially with position control where many direction switches occur.

The actuator synchronization requirement is dropped, because the negative effect on actuator synchronization is regarded as less important than the negative effect on performance.
4.5.2 Problem 2

The second problem is shown in figure 4.6 (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 4.6: Problem starting driving mode: Original problem (left) and solution (right)](image)

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.

```plaintext
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 4.6 (right).

4.5.3 Problem 3

The third problem is shown in figure 4.7 (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.

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 4.7 (right).

```plaintext
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 ) +
```
4.6 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.

4.6.1 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 B.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 $\phi$.

All actions, which are not used in the requirements, are hidden. After abstracting away from the internal behaviour, a reduced branching bisimilar 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 $\mu$-calculus. A short introduction to these different logics can be found in [MSS99]. Only modal $\mu$-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 $\mu$-calculus is used, which is developed for the model-checker $Evaluator^2$: regular alternation-free $\mu$-calculus. The syntax of regular alternation-free $\mu$-calculus is as follows [MS00]:

\[ (send_{Spv\_Mode}(i,modeMenu) \land eq(ua,userReset) \lor \delta ) \]

\[ (eq(e,errorNo) \lor send_{Spv\_Mode}(i,modeError) \cdot Supervisor\_Error(i) ) \]
The action formulae $\alpha$ are built from action names $a$ and the boolean operators ($\neg$), ($\lor$) and ($\land$). Furthermore the sign $\top$ is introduced to indicate the set of all actions. The regular formulae $\beta$ 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 $\phi$ 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 $\beta$ to a state where $\phi$ holds’ and $[\beta] \phi$ means ‘$\phi$ holds in all states reachable by a sequence of actions $\beta$’. 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 $\mu$-calculus. However working with fixpoint operators is very tricky and hard to understand. Therefore the ($^*$) is introduced in regular alternation-free $\mu$-calculus, which is more intuitive. The fixpoint operators and ($^*$) are related as follows:

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

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

4.6.2 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 $\alpha_{user}$ is to indicate all user actions.
  
  $\alpha_{user}(i) \equiv start(i) \lor stop(i) \lor emergency(i) \lor reset(i)$
  
  $\alpha_{user} \equiv \alpha_{user}(macs1) \lor \alpha_{user}(macs2)$

- The action formula $\alpha_{error}$ 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.
  
  $\alpha_{error}(i) \equiv$
  
  $\text{comm}_\text{Error}_\text{Set}(i, \text{errorSensor}) \lor$
  
  $\text{comm}_\text{Error}_\text{Set}(i, \text{errorMACS}) \lor$
  
  $\text{comm}_\text{Error}_\text{Set}(i, \text{errorAlive}) \lor$
  
  $\text{comm}_\text{Error}_\text{Set}(i, \text{errorController}) \lor$
  
  $\text{comm}_\text{Error}_\text{Set}(i, \text{errorActuator})$

- ...
4.6. Verification

\[ \alpha_{\text{error}} \equiv \alpha_{\text{error}}(\text{macs1}) \lor \alpha_{\text{error}}(\text{macs2}) \]

- The action formula \( \alpha_{\text{errorknown}} \) 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.

\[ \alpha_{\text{errorknown}}(i) \equiv \text{comm_Error_Get}(i, \text{errorSensor}) \lor \text{comm_Error_Get}(i, \text{errorMACS}) \lor \text{comm_Error_Get}(i, \text{errorAlive}) \lor \text{comm_Error_Get}(i, \text{errorController}) \lor \text{comm_Error_Get}(i, \text{errorActuator}) \]

\[ \alpha_{\text{errorknown}} \equiv \alpha_{\text{errorknown}}(\text{macs1}) \lor \alpha_{\text{errorknown}}(\text{macs2}) \]

- The action formula \( \alpha_{\text{mode}} \) is to indicate all mode actions.

\[ \alpha_{\text{mode}}(i) \equiv \text{menu}(i) \lor \text{drive}(i) \lor \text{error}(i) \]

\[ \alpha_{\text{mode}} \equiv \alpha_{\text{mode}}(\text{macs1}) \lor \alpha_{\text{mode}}(\text{macs2}) \]

Some system properties are expressed in regular formulae.

- The regular formula \( \beta_{\text{menu}} \) 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.

\[ \beta_{\text{menu}} \equiv \top^* \cdot (\text{menu}(\text{macs1}) \cdot (\top \land \neg \alpha_{\text{mode}}(\text{macs1}) \land \neg \alpha_{\text{error}}(\text{macs1}))^* \cdot \text{menu}(\text{macs2}) + \text{menu}(\text{macs2}) \cdot (\top \land \neg \alpha_{\text{mode}}(\text{macs2}) \land \neg \alpha_{\text{error}}(\text{macs2}))^* \cdot \text{menu}(\text{macs1})) \]

- The regular formula \( \beta_{\text{drive}} \) is to indicate each state reachable where both control systems are in driving mode.

\[ \beta_{\text{drive}} \equiv \top^* \cdot (\text{drive}(\text{macs1}) \cdot (\top \land \neg \alpha_{\text{mode}}(\text{macs1}) \land \neg \alpha_{\text{error}}(\text{macs1}))^* \cdot \text{drive}(\text{macs2}) + \text{drive}(\text{macs2}) \cdot (\top \land \neg \alpha_{\text{mode}}(\text{macs2}) \land \neg \alpha_{\text{error}}(\text{macs2}))^* \cdot \text{drive}(\text{macs1})) \]

- The regular formula \( \beta_{\text{error}} \) is to indicate each state reachable where both control systems are in error mode.

\[ \beta_{\text{error}} \equiv \top^* \cdot (\text{error}(\text{macs1}) \cdot (\top \land \neg \alpha_{\text{mode}}(\text{macs1}))^* \cdot \text{error}(\text{macs2}) + \text{error}(\text{macs2}) \cdot (\top \land \neg \alpha_{\text{mode}}(\text{macs2}))^* \cdot \text{error}(\text{macs1})) \]

The requirements, specified in section 4.3, 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 \textit{something bad will never happen} and the second expresses that \textit{something good can happen}.

- Deadlock freeness:

\[ [\top^* ](\top) \top \]
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:

\[ [β_{\text{drive}} \cdot α_{\text{other}}^* \cdot α_{\text{mode}}\backslash\{\text{drive}\}(i) ] F \]
\[ \land \]
\[ [β_{\text{drive}} ](α_{\text{other}}^* \cdot \text{drive}(i)) ] T \]

with \( α_{\text{other}} \equiv T \land \neg α_{\text{user}} \land \neg α_{\text{error}} \land \neg α_{\text{mode}} \)

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 \text{start}). The liveness part states that it is possible to stay in driving mode.

- Requirement II:

\[ [β_{\text{menu}} \cdot α_{\text{other}}^* \cdot \text{start}(i) \cdot α_{\text{other}}^* \cdot α_{\text{mode}}\backslash\{\text{drive}\}(i) ] F \]
\[ \land \]
\[ [β_{\text{menu}} \cdot α_{\text{other}}^* \cdot \text{start}(i) ] \]
\[ (α_{\text{other}}^* \cdot (\text{drive}(i) | (α_{\text{error}}^* \cdot α_{\text{other}} \cdot \text{menu}(i)))) ] T \]

with \( α_{\text{other}} \equiv T \land \neg α_{\text{user}} \land \neg α_{\text{error}} \land \neg α_{\text{mode}} \land \neg α_{\text{error}}^* \)

The safety part of the modal formula states that when a \text{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 C.2) also expresses that under the same circumstances both control systems go to driving mode after a \text{start} action.

- Requirement III:

\[ [β_{\text{drive}} \cdot α_{\text{other}}^* \cdot \text{stop}(i) \cdot α_{\text{other}}^* \cdot α_{\text{mode}}\backslash\{\text{menu}\}(i) ] F \]
\[ \land \]
\[ [β_{\text{drive}} \cdot α_{\text{other}}^* \cdot \text{stop}(i) ](α_{\text{other}}^* \cdot \text{menu}(i)) ] T \]

with \( α_{\text{other}} \equiv T \land \neg α_{\text{user}} \land \neg α_{\text{error}} \land \neg α_{\text{mode}} \)

The safety part of the modal formula states that when a \text{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 C.2) also expresses that under the same circumstances both control systems go to menu mode after a \text{stop} action.
4.6. Verification

• Requirement IV:

\[
\begin{align*}
\beta & \cdot \alpha & \cdot \text{emergency}(i) \cdot \alpha & \cdot \text{mode}(error)(i) \quad F \\
\beta & \cdot \alpha & \cdot \text{emergency}(i) \quad \langle \alpha & \cdot \text{error}(i) \rangle \quad T
\end{align*}
\]

with \( \alpha \equiv T \land \neg \alpha_{\text{user}} \land \neg \alpha_{\text{mode}} \)

The safety part of the modal formula states that when an 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 C.2) also expresses that under the same circumstances both control systems go to error mode after an emergency action.

• Requirement V:

\[
\begin{align*}
\beta & \cdot \alpha & \cdot \text{error}(i) \cdot \alpha & \cdot \text{mode}(error)(i) \quad F \\
\beta & \cdot \alpha & \cdot \text{error}(i) \quad \langle \alpha & \cdot \text{error}(i) \rangle \quad T
\end{align*}
\]

with \( \alpha \equiv T \land \neg \alpha_{\text{user}} \land \neg \alpha_{\text{error}} \land \neg \alpha_{\text{mode}} \)

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.

\[
\begin{align*}
\top & \cdot \text{menu}(macs1) \cdot \alpha & \cdot \text{mode}(macs2) \cdot \alpha & \cdot \text{drive}(macs1) \\
\beta & \cdot \text{autofsyc} \cdot \alpha & \cdot \text{menu}(macs1) \cdot \alpha & \cdot \text{mode}(macs2) \cdot \alpha & \cdot \text{drive}(macs1) \\
\beta & \cdot \text{insync} \cdot \alpha & \cdot \text{mode} \cdot \beta & \cdot \text{insync} \cdot \alpha & \cdot \text{mode} \cdot \beta & \cdot \text{insync} \cdot \alpha & \cdot \text{mode} \cdot \alpha & \cdot \text{menu}(macs2) \cdot \alpha & \cdot \text{menu}(macs2) \\
\end{align*}
\]

with \( \alpha \equiv T \land \neg \alpha_{\text{mode}} \)

\( \beta_{\text{autofsyc}} \equiv (\alpha \cdot \text{menu}(macs1) \cdot \alpha \cdot \text{drive}(macs2) \cdot \ldots) \)

\( \beta_{\text{insync}} \equiv (\alpha \cdot \text{menu}(macs1) \cdot \alpha \cdot \text{menu}(macs2) \cdot \ldots) \)
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 4.6.4) 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.

\[
\begin{align*}
\beta_{\text{drive}} \\
\cdot \alpha_{\text{other}} \\
\cdot \text{comm}_\text{SensorValue}_\text{Get}(i,\text{SensorValueL1}) \\
\cdot \text{setpoint}(i,\text{setpointValueR1}) \\
\cdot (\text{actuator}(i,\text{actuatorActionL1}) | \text{actuator}(i,\text{actuatorAction0})) \] F \\
\wedge \\
\begin{align*}
\beta_{\text{drive}} \\
\cdot \alpha_{\text{other}} \\
\cdot \text{comm}_\text{SensorValue}_\text{Get}(i,\text{SensorValueL1}) \\
\cdot \text{setpoint}(i,\text{setpointValueR1}) \\
\cdot \{ \text{actuator}(i,\text{actuatorActionR1}) \} T
\end{align*}
\]

with \( \alpha_{\text{other}} \equiv \top \wedge \neg \alpha_{\text{user}} \{ \text{start} \} \wedge \neg \alpha_{\text{error}} \wedge \neg \alpha_{\text{mode}} \)

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 \( \text{actuatorActionR1} \) and \( \text{actuatorActionL1} \).

\[
\begin{align*}
\beta_{\text{drive}} \\
\cdot \alpha_{\text{other}} ^* \cdot \text{drive}(\text{macs1}) \\
\cdot \alpha_{\text{other}} ^* \cdot \text{actuator}(\text{macs1},\text{actuatorActionR1}) \\
\cdot \alpha_{\text{other}} ^* \cdot \text{drive}(\text{macs2}) \\
\cdot \alpha_{\text{other}} ^* \cdot \text{actuator}(\text{macs2},\text{actuatorActionL1}) \] F \\
\wedge \\
\begin{align*}
\beta_{\text{drive}} \\
\cdot \alpha_{\text{other}} ^* \cdot \text{drive}(\text{macs1}) \\
\cdot \alpha_{\text{other}} ^* \cdot \text{actuator}(\text{macs1},\text{actuatorActionR1}) \\
\cdot \alpha_{\text{other}} ^* \cdot \text{drive}(\text{macs2}) \\
\cdot \{ \alpha_{\text{other}} ^* \} \\
\cdot \{ \text{actuator}(i,\text{actuatorActionR1}) | \text{actuator}(i,\text{actuatorAction0}) \} \} T
\end{align*}
\]
with $\alpha_{\text{other}} \equiv T \land \neg \alpha_{\text{user}} \{\text{start}\} \land \neg \alpha_{\text{error}} \land \neg \alpha_{\text{mode}}$

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.

### 4.6.3 Generation transition system

To verify the control system specification, first the transition system needs to be generated. This is done by using the $\mu$CRL toolset (see appendix B.1). First the transition system of a single control system is generated. The transition system is visualized in figure 4.8 using $FSM\ Visualizer^3$.

![Figure 4.8: Single control system using FSM Visualizer](image)

$^3$3D Visualization tool to analyse transition systems (see appendix B.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 4.8.

NoodleView⁴ 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 4.9 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 4.9: Mode switching behaviour in NoodleView

⁴Visualization tool to analyse transition system focused on state variables (see appendix B.3)
4.6. Verification

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 4.4.3 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 4.10 using FSM Visualizer to show the complexity.

![Figure 4.10: Detail of complete control system](image)

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.

### 4.6.4 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 Cesar Aldebaran Development Package to generate communicating transition systems (see appendix B.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 will 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 4.1 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.

<table>
<thead>
<tr>
<th>Mode transition</th>
<th>Synchronization time</th>
</tr>
</thead>
<tbody>
<tr>
<td>menu $\rightarrow$ drive</td>
<td>6 time samples</td>
</tr>
<tr>
<td>menu $\rightarrow$ error</td>
<td>Immediately</td>
</tr>
<tr>
<td>drive $\rightarrow$ menu</td>
<td>4 time samples</td>
</tr>
<tr>
<td>drive $\rightarrow$ error</td>
<td>4 time samples</td>
</tr>
<tr>
<td>error $\rightarrow$ menu</td>
<td>Immediately</td>
</tr>
</tbody>
</table>

Table 4.1: 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.
4.7 Control system software in Simulink

The software for the control system is built in $\mu$CRL to be able to validate the software design. However $\mu$CRL code cannot be compiled to run on a control system directly. This means the $\mu$CRL code has to be converted to another programming environment first. Since it is only possible to use Simulink to generate code for the MACS control system, it needs to be converted to a Simulink model.

To preserve validity of the software, a formal relation must be made between Simulink and $\mu$CRL 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 $\mu$CRL is a process-algebraic programming language to design and analyze communicating processes with data.

The $\mu$CRL code cannot 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 $\mu$CRL 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 $\mu$CRL code.

The general composition of the Simulink model is shown in figure 4.11. 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 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 $\mu$CRL model, the controller is simplified to reduce the complexity of the state space (to avoid state explosion) without losing the characteristics of a controller. This simplified controller needs to be replaced by two PID controllers.

The controller process is shown in figure 4.12. The processes controlvoter and controlswitch are used to choose the right control strategy: no control, torque control or position control.

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.

---

6This process also includes monitoring the other control system, unlike the $\mu$CRL model.
Safety-critical software design

Figure 4.11: Control system model in Simulink

Figure 4.12: Controller process in Simulink
Chapter 5

Driving tests and safety

‘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. But first a simulation model needs to be built in Simulink. The model consists of three parts (figure 5.1): the control system (section 5.1), the robot dynamics (section 5.2) and the vehicle model (section 5.3). In this chapter the focus will be on the overall behaviour of the steering robot in representative driving tests.

Figure 5.1: Simulation model for steering robot

5.1 Control system

The software for the control system is developed in chapter 4. For the simulation also the behaviour of the control system needs to be implemented. Otherwise effects like sample frequency and computation delay 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
Driving tests and safety

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 estimated at half the sample time, since the control system runs at the highest possible sample frequency without failing to fulfill all computations.

The effects of sample frequency and computation delay in the control system model are shown in figure 5.2. 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.

\[\text{Figure 5.2: Step torque by the control system}\]

\(^1\)The torque is computed from the three phase currents and the actual angle. The computation will be further explained in section 5.2
5.2 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.

5.2.1 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.

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 5.3. Because the steering wheel angle is fixed and the sensor is located in the motor, the sensor measures the angle ($\alpha$) between steering wheel hub and motor instead of the steering wheel angle. This situation can be modeled as shown in figure 5.4.

The resulting differential equation is described as follows.
Driving tests and safety

\[ J_{gda} \ddot{\alpha} = T_{gda} - T_d - T_k \]
\[ \iff \]
\[ J_{gda} \ddot{\alpha} + d_{gda} \dot{\alpha} + k_{gda} \alpha = T_{gda} \] (5.1)

5.2.2 Friction

The friction of the system is determined experimentally by simply measuring the minimal forces needed to change the steering wheel angle.

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 \( T_{\text{fric}} \) 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.

5.2.3 Backlash

Backlash\(^2\) can be measured by applying positive and negative torques to the steering robot, when it is fixated. 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 experiment are shown in figure 5.5.

Unfortunately other effects are measured as well, like hysteresis and friction (equilibrium between friction and flexibility). However the effects are expected to be low. Hysteresis effects can be kept small by applying small forces to the steering robot. The effect of friction is expected to be small, since friction is also small. Backlash \( \alpha_{\text{backlash}} \) is therefore estimated at 0.2 degrees.

\( ^2\)Backlash is the mechanical term for loss of motion due to clearance between mechanical components. [WIK]
5.2.4 Fit parameters

The second order model can now be extended with friction and backlash. The model is shown in figure 5.6.

![Second order system with non linear effects](image)

Figure 5.6: 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.

\[
J_{gda} \ddot{\alpha} = T_{gda} - T_d - T_k - T_{fric}
\]  

(5.2)

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 static.

5.2.5 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.
Driving tests and safety

\[
J_{gda} \ddot{\alpha}_{gda} = T_{gda} - T_d - T_k - T_{fric}
\]
\[
\iff\quad J_{gda} \ddot{\alpha}_{gda} + d_{gda}(\alpha_{gda} - \dot{\alpha}_{st}) + k_{gda}(\alpha_{gda} - \alpha_{st}) + T_{fric} = T_{gda}
\]

with \( T_{gda} = i_{st} T_m \) and \( \alpha_{gda} = \alpha_m / i_{st} \)

The relation between actual steering torque and motor torque needs to be adjusted to include the dynamic behaviour of the steering robot. By rewriting (5.3) the actual steering torque can now be described by (5.4).

\[
T_{st}(= T_d + T_k) = T_{gda} - J_{gda} \ddot{\alpha}_{gda} - T_{fric}
\]

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 \( \alpha_{gda} \) is the only unknown variable.

The Simulink model can now be implemented. It is shown in figure 5.7.

\[\text{Figure 5.7: Dynamics in Simulink}\]

5.3 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.

5.3.1 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.

5.3.2 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.

5.3.3 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 5.8. 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 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 2.4. 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
Driving tests and safety

Figure 5.8: Model of steering mechanism

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 5.9 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 5.9: The effect of power steering on maximum steering speed
5.4 Simulation

Until now overall safety is realized by validating the software requirements and by duplicating the hardware to make it fault-tolerant. However, 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 the error mode strategy for failing hardware is determined and second for an emergency.

5.4.1 Hardware failure strategy

First the effect of failing hardware during driving tests is analyzed. Because the steering robot is the most sensitive to hardware failure, a driving test is needed with high demands on steering speed. The ISO 3888 double lane change driving test is chosen with the developed vehicle model at 60 km/h. Angle setpoints are determined experimentally which let the vehicle drive through the desired trajectory. Figure 5.10 shows the double lane change with normal fault-tolerant control.

![Double lane change at 60 kph](image)

In case of a failing hardware component, other components try to fulfill the tasks. If for example an actuator fails the other actuator tries to double its efforts within its specification. Unfortunately some performance is lost. Figure 5.11 shows the effect of having a single actuator on the double lane change driving test after 1.4 seconds. Although the prescribed trajectory can no longer be fulfilled with a single actuator, the followed trajectory is acceptable.

It is also an option to stop the vehicle if a hardware component fails. However chances are that the vehicle gets in an uncontrollable slip. 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 5.12. The vehicle clearly becomes uncontrollable.

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 trajec as the braking of figure 5.12.

The best open-loop strategy during hardware failure is to try to complete the trajectory with reduced performance. It is better to deviate slightly from the desired trajectory than to cause the vehicle to become uncontrollable. 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.

## 5.4.2 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 an uncontrollable slip is inevitable.

![Figure 5.13: Area needed for double lane change at 60 kph](image)

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. The driving test is simulated with emergencies at different times in figure 5.13. The figure gives an indication for the area needed for a double lane change driving test.
Chapter 6
Conclusions & Recommendations

'Reality is both continuous and discrete.'
Geoff Hasellhurst

6.1 Conclusions
The conclusions are split up in four parts. First conclusions are given on hardware design, software design and vehicle dynamics followed by some general conclusions.

6.1.1 Hardware 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 was 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.

The dual control setup is chosen for the control system, where each actuator has its own controller. The performance in the control gap is superior and no additional hardware is needed to implement this. This setup reduces hardware complexity but increases software complexity due to the synchronization problems.
6.1.2 Software design

The development of the control system software had little problems. The use of formal methods certainly proved to be a good way to manage the complexity. The requirements of the control system could be verified in µCRL and the implementation of the model in Simulink immediately worked.

During the verification of the model problems with the original software design were found, which were almost impossible to detect without formal methods. Sequences of user actions were discovered, which conflicted with the requirements. Furthermore conflicting requirements were found. Although the basic functionality of the control system is fairly simple, the hardware redundancy introduced 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 could deteriorate software safety. In this case the GDA became 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 was too large for the tools and some tricks are used to reduce its size. First the 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.

6.1.3 Vehicle dynamics

Since the GDA was not operational on time, driving tests could only be done with simulations. A simulation model is built in Simulink which includes the control system, the steering robot dynamics and a vehicle model. It is used to find out what the best strategy is for the error mode.

The safety during driving tests can not be verified formally, simulations are used instead. Based on the results of simulations with a double lane change driving test, the best strategy during hardware failure turned out to be to complete the trajectory with reduced performance. 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.
6.2. Recommendations

With the error mode designed, an area is indicated which is needed for the double lane change 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.

6.1.4 Project

Combining both fields in an early phase improved the design process. Some hardware designs to improve safety compromised the safety of the software, because it introduced the need for synchronization, while other hardware designs needed difficult hardware components. By looking at both aspects early in the design process, a better understanding of all possible options is obtained, which ultimately leads to a better decision.

6.2 Recommendations

The simulation needs to be validated on a test track in reality. 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 7

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. \( \chi \) [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 report an artificial split is made between discrete and continuous behaviour to be able to verify the requirements individually. \( \mu \text{CRL} \) 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


[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)


Appendix A

Transition system reduction using binary semaphores

A.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.

A.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 A.1.

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.

\[
\text{sort SemaphoreAction} \\
\text{func P,V: SemaphoreAction} \\
\text{act comm Semaphore: SemaphoreAction} \\
\text{send Semaphore: SemaphoreAction} \\
\text{recv Semaphore: SemaphoreAction} \\
\text{comm send Semaphore | recv Semaphore = comm Semaphore}
\]
A.2. Technique

```plaintext
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 A.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 A.2: Direct communication](image)

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 A.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 A.3: Loss of behaviour: Problem (left) and solution (right)](image)


A.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.

A.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.

The canbus is a delayed buffer. One process 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 separated 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 A.4.

![Figure A.4: Steering robot](image)

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.

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 B

Process algebraic tools

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

B.1 $\mu$CRL toolset

Most tools used are in the $\mu$CRL toolset developed at CWI\(^1\). The following tools are part of this toolset:

- **Mcrl**: This tool checks if the $\mu$CRL specification is well-formed. It can also transform $\mu$CRL 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 $\mu$CRL 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 $\mu$CRL elements. Several reduction tools are included: \texttt{Rewr}, \texttt{ConstElm}, \texttt{ParElm}, \texttt{SumElm} and \texttt{StructElm}.

- **Confcheck**: This tool tries to reduce the LPO by confluent $\tau$ 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 B.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
Process algebraic tools

Figure B.1: μCRL toolset dependence

B.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.

B.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 C

Formal methods code

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

C.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
Formal methods code

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,
Formal methods code

**errorController:** $\rightarrow$ Error

map $\text{eq:Error#Error} : \rightarrow \text{Bool}$

rew $\text{eq(errorNo, errorNo)} = \text{T}$  
$\text{eq(errorNo, errorMACS)} = \text{F}$  
$\text{eq(errorNo, errorActuator)} = \text{F}$  
$\text{eq(errorNo, errorSensor)} = \text{F}$  
$\text{eq(errorNo, errorController)} = \text{F}$  
$\text{eq(errorMACS, errorNo)} = \text{F}$  
$\text{eq(errorMACS, errorMACS)} = \text{T}$  
$\text{eq(errorMACS, errorActuator)} = \text{F}$  
$\text{eq(errorMACS, errorSensor)} = \text{F}$  
$\text{eq(errorMACS, errorController)} = \text{F}$  
$\text{eq(errorActuator, errorNo)} = \text{F}$  
$\text{eq(errorActuator, errorMACS)} = \text{F}$  
$\text{eq(errorActuator, errorActuator)} = \text{T}$  
$\text{eq(errorActuator, errorSensor)} = \text{F}$  
$\text{eq(errorActuator, errorController)} = \text{F}$  
$\text{eq(errorSensor, errorNo)} = \text{F}$  
$\text{eq(errorSensor, errorMACS)} = \text{F}$  
$\text{eq(errorSensor, errorActuator)} = \text{F}$  
$\text{eq(errorSensor, errorSensor)} = \text{T}$  
$\text{eq(errorSensor, errorController)} = \text{F}$  
$\text{eq(errorController, errorNo)} = \text{F}$  
$\text{eq(errorController, errorMACS)} = \text{F}$  
$\text{eq(errorController, errorActuator)} = \text{F}$  
$\text{eq(errorController, errorSensor)} = \text{F}$  
$\text{eq(errorController, errorController)} = \text{T}$

%======================================================================% SetpointValue type %======================================================================% sort SetpointValue %======================================================================% func $\text{setpointValueL1, setpointValue0, setpointValueR1} : \rightarrow \text{SetpointValue}$ %======================================================================% map $\text{eq:SetpointValue#SetpointValue} : \rightarrow \text{Bool}$ %======================================================================% rew $\text{eq(setpointValueL1, setpointValueL1)} = \text{T}$  
$\text{eq(setpointValueL1, setpointValueL1)} = \text{F}$  
$\text{eq(setpointValueL1, setpointValueR1)} = \text{F}$  
$\text{eq(setpointValue0, setpointValueL1)} = \text{T}$  
$\text{eq(setpointValue0, setpointValueL1)} = \text{F}$  
$\text{eq(setpointValue0, setpointValueR1)} = \text{F}$  
$\text{eq(setpointValue0, setpointValue0)} = \text{F}$  
$\text{eq(setpointValueR1, setpointValueL1)} = \text{F}$  
$\text{eq(setpointValueR1, setpointValue0)} = \text{F}$  
$\text{eq(setpointValueR1, setpointValue0)} = \text{T}$

%======================================================================% 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) = F
  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
Formal methods code

\[
\begin{align*}
\min(\text{sensorAngle}_0, \text{sensorAngle}_0) &= \text{sensorAngle}_0 \\
\min(\text{sensorAngle}_0, \text{sensorAngle}_R1) &= \text{sensorAngle}_0 \\
\min(\text{sensorAngle}_R1, \text{sensorAngle}_L1) &= \text{sensorAngle}_L1 \\
\min(\text{sensorAngle}_R1, \text{sensorAngle}_0) &= \text{sensorAngle}_0 \\
\min(\text{sensorAngle}_R1, \text{sensorAngle}_R1) &= \text{sensorAngle}_R1
\end{align*}
\]

\[
\text{map max:SensorAngle#SensorAngle} \rightarrow \text{SensorAngle}
\]

\[
\begin{align*}
\text{rew} \quad \max(\text{sensorAngle}_L1, \text{sensorAngle}_L1) &= \text{sensorAngle}_L1 \\
\max(\text{sensorAngle}_L1, \text{sensorAngle}_0) &= \text{sensorAngle}_0 \\
\max(\text{sensorAngle}_L1, \text{sensorAngle}_R1) &= \text{sensorAngle}_R1 \\
\max(\text{sensorAngle}_0, \text{sensorAngle}_L1) &= \text{sensorAngle}_0 \\
\max(\text{sensorAngle}_0, \text{sensorAngle}_0) &= \text{sensorAngle}_0 \\
\max(\text{sensorAngle}_0, \text{sensorAngle}_R1) &= \text{sensorAngle}_R1 \\
\max(\text{sensorAngle}_R1, \text{sensorAngle}_L1) &= \text{sensorAngle}_R1 \\
\max(\text{sensorAngle}_R1, \text{sensorAngle}_0) &= \text{sensorAngle}_R1 \\
\max(\text{sensorAngle}_R1, \text{sensorAngle}_R1) &= \text{sensorAngle}_R1
\end{align*}
\]

\[
\text{map median:SensorAngle#SensorAngle#SensorAngle} \rightarrow \text{SensorAngle}
\]

\[
\begin{align*}
\text{var} \quad \text{sv1, sv2, sv3: SensorAngle} \\
\text{rew} \quad \text{median}(\text{sv1, sv2, sv3}) &= \max( \max( \min(\text{sv1, sv2}), \min(\text{sv1, sv3}) ), \min(\text{sv2, sv3}) )
\end{align*}
\]

\[
\text{map fault:SensorAngle#SensorAngle#SensorAngle#SensorAngle} \rightarrow \text{Bool}
\]

\[
\begin{align*}
\text{var} \quad \text{svMedian, sv1, sv2, sv3: SensorAngle} \\
\text{rew} \quad \text{fault}(\text{svMedian, sv1, sv2, sv3}) &= \not( \text{and}( \text{and}( \text{match}(\text{svMedian, sv1}), \text{match}(\text{svMedian, sv2}) ), \text{match}(\text{svMedian, sv3}) ) )
\end{align*}
\]

\%
\% ActuatorAction type
\%

\[
\text{sort ActuatorAction}
\]

\[
\text{func} \quad \text{actuatorAction}_L1, \text{actuatorAction}_0, \text{actuatorAction}_R1, \text{actuatorActionError} :\rightarrow \text{ActuatorAction}
\]

\[
\text{map eq:ActuatorAction#ActuatorAction} \rightarrow \text{Bool}
\]

\[
\begin{align*}
\text{rew} \quad \text{eq}(\text{actuatorAction}_L1, \text{actuatorAction}_L1) &= \text{T} \\
\text{eq}(\text{actuatorAction}_L1, \text{actuatorAction}_0) &= \text{F} \\
\text{eq}(\text{actuatorAction}_L1, \text{actuatorAction}_R1) &= \text{F} \\
\text{eq}(\text{actuatorAction}_L1, \text{actuatorActionError}) &= \text{F} \\
\text{eq}(\text{actuatorAction}_0, \text{actuatorAction}_L1) &= \text{F} \\
\text{eq}(\text{actuatorAction}_0, \text{actuatorAction}_0) &= \text{T} \\
\text{eq}(\text{actuatorAction}_0, \text{actuatorAction}_R1) &= \text{F} \\
\text{eq}(\text{actuatorAction}_0, \text{actuatorActionError}) &= \text{F} \\
\text{eq}(\text{actuatorAction}_R1, \text{actuatorAction}_L1) &= \text{F} \\
\text{eq}(\text{actuatorAction}_R1, \text{actuatorAction}_0) &= \text{F} \\
\text{eq}(\text{actuatorAction}_R1, \text{actuatorAction}_R1) &= \text{T} \\
\text{eq}(\text{actuatorAction}_R1, \text{actuatorActionError}) &= \text{F} \\
\text{eq}(\text{actuatorActionError}, \text{actuatorAction}_L1) &= \text{F} \\
\text{eq}(\text{actuatorActionError}, \text{actuatorAction}_0) &= \text{F}
\end{align*}
\]

- 80 -
C.1. Specification

\begin{align*}
eq(\text{actuatorActionError,actuatorActionR1}) &= F \\
eq(\text{actuatorActionError,actuatorActionError}) &= T
\end{align*}

% 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
% Semaphore Process
% Time actions
% Time processes

\begin{center}
%......................................................
% action
%......................................................
\end{center}
Formal methods code

% 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
Formal methods code

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
    send_Mon_Alive: MacsIndex # Bool
    recv_Mon_Alive: MacsIndex # Bool

- 84 -
C.1. Specification

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)
  )

  - 85 -
Formal methods code

\[
\begin{align*}
\text{proc Monitor\_Out}(i: \text{MacsIndex}) = \\
&\quad \text{send\_Mon\_Alive}(i, T) \cdot \text{Monitor\_Out\_Mode}(i) \\
\text{proc Monitor\_Out\_Mode}(i: \text{MacsIndex}) = \\
&\quad \text{sum}(e: \text{Error}, \text{recv\_Mon\_Error}(i, e) \cdot \\
&\quad \quad \text{sum}(m: \text{Mode}, \text{recv\_Mon\_Mode}(i, m) \cdot \\
&\quad \quad \quad (\begin{align*}
&\quad \quad \quad \quad (\text{send\_Mon\_Mode\_Other}(i, \text{otherMenu}) \\
&\quad \quad \quad \quad \quad \quad (<|\text{eq}(e, \text{errorNo})|> \\
&\quad \quad \quad \quad \quad \quad \text{send\_Mon\_Mode\_Other}(i, \text{otherMenuError}) \\
&\quad \quad \quad \quad \quad \quad <|\text{eq}(m, \text{modeMenu})|> \text{delta} \\
&\quad \quad \quad \quad ) + \\
&\quad \quad \quad \quad (\text{send\_Mon\_Mode\_Other}(i, \text{otherDrive}) \\
&\quad \quad \quad \quad \quad \quad <|\text{eq}(m, \text{modeDrive})|> \text{delta} \\
&\quad \quad \quad \quad ) + \\
&\quad \quad \quad \quad (\text{send\_Mon\_Mode\_Other}(i, \text{otherError}) \\
&\quad \quad \quad \quad \quad \quad <|\text{eq}(m, \text{modeError})|> \text{delta} \\
&\quad \quad \quad \quad ) \\
&\end{align*}) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{proc Angle}(i: \text{MacsIndex}) = \\
&\quad \text{sum}(sv1: \text{SensorAngle}, \text{recv\_Angle\_Sensor}(i, \text{sensor1}, sv1) \cdot \\
&\quad \quad \text{sum}(sv2: \text{SensorAngle}, \text{recv\_Angle\_Sensor}(i, \text{sensor2}, sv2) \cdot \\
&\quad \quad \quad \text{sum}(sv3: \text{SensorAngle}, \text{recv\_Angle\_Sensor}(i, \text{sensor3}, sv3) \cdot \\
&\quad \quad \quad \quad \text{send\_Angle\_SensorValue}(i, \text{median}(sv1, sv2, sv3)) \cdot \\
&\quad \quad \quad \quad \text{send\_Angle\_SensorError}(i, \text{fault}(\text{median}(sv1, sv2, sv3), sv1, sv2, sv3)) \\
&\end{align*}
\]

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

act error\_Message: \text{MacsIndex} \# Error

\%======================================================================
\% Supervisor processes
\%
\% Supervisor takes care of:
\% \quad \text{-System check}
\% \quad \text{-Controller mode selection}

- 86 -
C.1. Specification

%====================================================================
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) =

- 87 -
Formal methods code

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,actuatorAction)|>
      ( 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).

  - 89 -
Formal methods code

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
)
C.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.

C.2.1 Requirement 2

\[
\begin{align*}
&\beta_{\text{menu}} \cdot \alpha_{\text{other}}^* \cdot \text{start}(i) \cdot \alpha_{\text{other}}^* \cdot \alpha_{\text{mode}\{\text{drive}\}(i)} F \\
\wedge &\beta_{\text{menu}} \cdot \alpha_{\text{other}}^* \cdot \text{start}(i) \\
&\langle \alpha_{\text{other}}^* \cdot \{ \text{drive}(i) | (\alpha_{\text{error\ known}} \cdot \alpha_{\text{other}} \cdot \text{menu}(i)) \} \rangle T \\
\wedge &\beta_{\text{menu}} \cdot \\
&\langle \alpha_{\text{other}}^* \cdot \text{menu}(i) \rangle T \\
\wedge &\beta_{\text{menu}} \\
&\alpha_{\text{other}} \cdot \alpha_{\text{other}}^* \cdot \text{drive}(i) + \cdot \\
&\alpha_{\text{other}} \cdot \text{menu}(i) \rangle T \\
\end{align*}
\]

with \( \alpha_{\text{other}} \equiv \top \wedge \neg \alpha_{\text{user}} \wedge \neg \alpha_{\text{error}} \wedge \neg \alpha_{\text{mode}} \wedge \neg \alpha_{\text{error\ known}} \)

C.2.2 Requirement 3

\[
\begin{align*}
&\beta_{\text{drive}} \cdot \alpha_{\text{other}}^* \cdot \text{stop}(i) \cdot \alpha_{\text{other}}^* \cdot \alpha_{\text{mode}\{\text{menu}\}(i)} F \\
\wedge &\beta_{\text{drive}} \cdot \alpha_{\text{other}}^* \cdot \text{stop}(i) \rangle \alpha_{\text{other}} \cdot \text{menu}(i) \rangle T \\
\wedge &\beta_{\text{drive}} \\
&\langle \alpha_{\text{other}}^* \cdot \text{menu}(i) \rangle T \\
\wedge &\beta_{\text{drive}} \\
&\alpha_{\text{other}} \cdot \text{stop}(i) \cdot \alpha_{\text{other}}^* \cdot \text{menu}(i) + \\
&\alpha_{\text{other}} \cdot \text{stop}(i) \cdot \alpha_{\text{other}} \cdot \alpha_{\text{mode}\{\text{menu}\}(i)} F \\
\wedge &\beta_{\text{drive}} \\
&\alpha_{\text{other}} \cdot \text{stop}(i) \cdot \alpha_{\text{other}}^* \cdot \text{menu}(i) + \\
&\alpha_{\text{other}} \cdot \text{menu}(i) \rangle T \\
\end{align*}
\]

with \( \alpha_{\text{other}} \equiv \top \wedge \neg \alpha_{\text{user}} \wedge \neg \alpha_{\text{error}} \wedge \neg \alpha_{\text{mode}} \)
C.2.3 Requirement 4

\[
\begin{align*}
& [ \beta_{drive} \cdot \alpha_{other}^* \cdot \text{emergency}(i) \cdot \alpha_{other}^* \cdot \alpha_{mode} \{ \text{error} \}(i) ] \ F \\
& \land \\
& [ \beta_{drive} \cdot \alpha_{other}^* \cdot \text{emergency}(i) ] \{ \alpha_{other}^* \cdot \text{error}(i) \} \ T \\
& \land \\
& [ \beta_{drive} \cdot \\
& (\alpha_{other}^* \cdot \text{emergency}(i) \cdot \alpha_{other}^* \cdot \text{error}(i))^+ \cdot \\
& \alpha_{other}^* \cdot \text{emergency}(other(i)) \cdot \alpha_{other}^* \cdot \alpha_{mode} \{ \text{error} \}(other(i)) ] \ F \\
& \land \\
& [ \beta_{drive} \cdot \\
& (\alpha_{other}^* \cdot \text{emergency}(i) \cdot \alpha_{other}^* \cdot \text{error}(i))^+ \cdot \\
& \alpha_{other}^* \cdot \text{emergency}(other(i))] \\
& \langle \alpha_{other}^* \cdot \text{error}(other(i)) \rangle \ T \\
\end{align*}
\]

with \( \alpha_{other} \equiv \top \land \neg \alpha_{user} \land \neg \alpha_{mode} \)