Design, Measurement and Verification of Scania’s Platform Software Architecture for Safety Related Embedded Systems

MARTIN HÄRBERG
ROBERTO CHIARITO

Master of Science Thesis
Stockholm, Sweden 2013
Design, Measurement and Verification of Scania’s Platform Software Architecture for Safety Related Embedded Systems

MARTIN HÄRBERG, ROBERTO CHIARITO

Master of Science Thesis MMK 2013:50 MES 003
KTH Industrial Engineering and Management
Machine Design
SE-100 44 STOCKHOLM
Abstract

The platform software architecture for the safety related embedded systems developed by Scania has become increasingly more complex. High complexity raises both the risk of failures and the time consumed by software developers to understand and debug the source code. This leads to increased software maintenance costs, which according to [24] can be between 60% and 75% of the total cost of software development.

The purpose of this Master’s thesis is to investigate how a part of Scania’s current software architectural design can be further developed in order to decrease the complexity and the maintenance costs, without compromising with the essential functionality and performance. Another goal is to provide a solution that complies with the software safety requirements from ISO 26262, which Scania is planning to be able to fulfill in the future.

To be able to compare our proposal for the software architecture with Scania’s current solution, a measurement tool has been developed. This tool measures the software quality metrics coupling and cohesion, which together with other software metrics gives an estimation of the architecture’s complexity. The verification of the software architecture with regards to ISO 26262 has been done using contract theory.

The thesis work has resulted in alternative solutions for the software architectural design of the pressure sensor driver and the real-time database in one of Scania’s electronic control units. These solutions comply better with ISO 26262 and have lower complexity than Scania’s current solution in terms of coupling, cohesion and size of software components. This has been achieved by restructuring the software architecture and avoiding reuse of common software functions. The main conclusion of the thesis is that there is great potential for Scania to reduce the complexity of the platform software architecture and comply with ISO 26262.
Sammanfattning

Plattformsarkitekturen för programvaran i de säkerhetsrelaterade inbyggda system som Scania utvecklar har blivit alltmer komplex. Hög komplexitet medför ökad risk för att fel uppstår i programvaran samt att den tid som programvaruutvecklare spenderar med att förstå och debugga (avlusa) källkoden ökar. Detta leder till ökade underhållskostnader, vilket enligt [24] kan utgöra mellan 60 % och 75 % av den totala kostnaden för programvaruutveckling.

Syftet med detta examensarbete är att undersöka hur en del av Scanias nuvarande arkitekturdesign kan vidareutvecklas för att minska komplexiteten, utan att kompromissa med någon grundläggande funktionalitet och prestanda. Ett annat mål är att erbjuda en lösning som uppfyller de säkerhetskrav för programvaran som ISO 26262 ställer, vilket Scania förbereder sig för att kunna uppfylla i framtiden.

Ett mätverktyg har utvecklats för att kunna jämföra vår programvaruarkitekturlösning med Scanias nuvarande lösning. Detta verktyg mäter kvalitetsmåten coupling (koppling) och cohesion (samhörighet), vilka tillsammans med andra programvarumått ger en uppskattning av komplexiteten för arkitekturen. Verifieringen av programvaruarkitekturen med avseende på kraven från ISO 26262 har utförts med hjälp av kontraktteori.

Examensarbetet har resulterat i alternativa arkitekturlösningar för trycksensorernas drivrutiner samt realtidsdatabasen i en av Scanias styreheter, där lösningarna både uppfyller kraven från ISO 26262 bättre och har lägre komplexitet än Scanias nuvarande lösning. Detta har uppnåtts genom en omstrukturering av programvaruarkitekturen samt genom att undvika att återanvända gemensamma programvarufunktioner. Huvudsatsen som kan dras från examensarbetet är att det finns stor potential för Scania att kunna reducera programvaruarkitekturons komplexitet, samt uppfylla kraven från ISO 26262.
Preface

There is an ongoing discussion today about whether or not to facilitate reuse of common software functions in a system’s software architecture. This topic and the consequences it has on safety related embedded systems is the main discussion in this Master’s thesis.

The thesis is the culmination of the Master of Science degree in Electrical Engineering at KTH Royal Institute of Technology. The thesis work is divided into three main parts, which are presented in chapters 4 to 6. Chapter 4, which presents the software architectural design, is written by Roberto Chiarito. Chapters 5 and 6, which presents a measurement tool and the verification of the software architecture, is written by Martin Härberg. The residual of the report is written by the authors together.

We would like to thank Scania for the opportunity to perform this Master’s thesis and their extensive support throughout the thesis. It has been a great privilege for us to work at Scania and the environment has been very friendly and rewarding. We would personally like to thank our supervisors Mattias Nyberg at Scania and Matthias Biehl at KTH Royal Institute of Technology for their support and good advice with the technical execution of the thesis, as well as the presentation of the thesis work. Finally, a special thanks to our examiner Martin Törngren who has given us valuable suggestions for improving the thesis.

NOTE TO THE READER: Due to the nature of the content in this Master’s thesis, all sensitive industrial details have been excluded from this version of the thesis.

Martin Härberg and Roberto Chiarito

Stockholm, June 2013
Nomenclature

Here are the abbreviations used in the thesis report listed.

**Abbreviations**

A/D  Analog/Digital  
APPL Application layer  
ASIL Automotive Safety Integrity Level  
BSW Basic Software  
CAN Controller Area Network  
COMP Common Platform  
CPU Central Processing Unit  
DIST Disturbed  
DPF Diesel Particulate Filter  
DTD Document Type Definition  
DTC Diagnostic Trouble Code  
E/E Electrical and Electronic  
ECU Electronic Control Unit  
EEC3 3rd generation Exhaust Emission Control system  
FILE File handler layer  
I/O Input/Output  
ISO International Organization for Standardization  
LLAP Low Level Application layer  
NOM Nominal  
NOx Nitrogen Oxide  
OBD On-Board Diagnosis  
QM Quality Management  
RTDB Real-Time Database  
SCR Selective Catalytic Reduction  
SW Software  
SYSM System Manager  
UNA Unavailable  
UTLS Utilities layer  
VSEN Virtual Sensor layer
## Contents

1 **Introduction**  
   1.1 Background  
   1.2 Challenges  
   1.3 Purpose  
   1.4 Delimitations  
   1.5 Methodology  
   1.6 Summary of the Results and Conclusions

2 **Frame-of-Reference**  
   2.1 Scania’s Architectural Design of the Platform Software  
   2.2 Fault-Tolerant Control of Electrical Control Units  
   2.3 Requirements for Product Development at the Software Level from ISO 26262

3 **Analysis**  
   3.1 Gap Analysis between ISO 26262 and Scania’s Current Software Architectural Design in EEC3  
   3.2 Gap Analysis between ISO 26262 and Scania’s Current Software Unit Design in EEC3

4 **Software Architectural Design**  
   4.1 Design and Implementation of the Pressure Sensor Driver in EEC3  
   4.2 Prototype Concepts for the Real-Time Database in EEC3

5 **Measurement Tool**  
   5.1 Measuring Complexity of a Software Architectural Design  
   5.2 Testing Procedure for the Measurement Tool

6 **Verification of the SW Architecture**  
   6.1 Verifying the Software Architectural Design for the Pressure Sensor Driver in EEC3  
   6.2 Verifying Consistency of the Software Safety Requirements

7 **Results**  
   7.1 Measurement Results of the Pressure Sensor Driver Implementations  
   7.2 Estimated Characteristics for the Real-Time Database Concepts
1. Introduction

This chapter describes the background, challenges, purpose, delimitations, and methodology used in the thesis work. A short summary of the results and conclusions is also included in this chapter.

1.1 Background

Scania is a world leading company in the manufacturing of heavy trucks and buses. They also manufacture diesel engines for heavy vehicles and general industrial applications. The trucks and buses that Scania develops have several Electronic Control Units (ECUs), which controls and manages the electrical system of the vehicle. An ECU is an embedded system, which consists of a circuit board with electronic components, such as a micro-controller, external memory systems, communication circuits and connectors for sensors and actuators. The software residing in the micro-controller is referred to as the platform software, and manages the whole ECU.

A large part of the ECUs is developed in-house by Scania, especially the platform software for the micro-controller. The platform software has functionality similar to a real-time operating system, where parts of it receives input signals from sensors and processes them so that they can be used by different software applications. The platform software is also responsible for performing diagnostics on the signals so that fault symptoms can be detected and handled. It is designed to be fault-tolerant which enables the ECU to operate with the same or reduced functionality even while a fault is present.

To reduce the risks of failures in the ECUs, an international standard regarding functional safety for road vehicles, ISO 26262, has been developed by the International Organization for Standardization [11]. As of now ISO 26262 is not legally required for trucks and buses, but Scania’s aim is to be prepared to meet the requirements of this standard in the future. ISO 26262 covers the whole development process of the different items of the vehicle; from specification of the items to their production release.

1.2 Challenges

The software architecture denotes the high level structures of the platform software. The architectural design of Scania’s platform software in their trucks and buses is unfortunately not compliant with all of the requirements from ISO 26262. Changes in the software architectural design are needed as well as new documentation processes for the software development.
In order to avoid failures resulting from high complexity, ISO 26262 requires the software architectural design to exhibit modularity, encapsulation and simplicity. Reducing the complexity of the source code and its architectural design is beneficial in many aspects. Apart from reducing the risk of failures, the time consumed for the software developers to understand the source code will be decreased. This leads to reduced maintenance costs for Scania, which according to [24] can be between 60% and 75% of the total cost of software development. It will also be easier for different software developers to collaborate with each other during the development and debugging process if the complexity is reduced.

Another requirement from ISO 26262 is that the software architectural design shall be verifiable, which means that a bi-directional traceability between the software architectural design and the software safety requirements is needed. The software safety requirements shall be allocated to the software components in the system.

1.3 Purpose

The purpose of the thesis is to investigate how Scania’s current architectural design of the platform software can be further developed in order to decrease the complexity, while fulfilling the safety requirements from ISO 26262. Complexity is measured by the software quality metrics coupling, cohesion and size of software components.

In order to propose a solution to the software developers at Scania, an objective measurement method needs to be developed, which can measure the complexity of Scania’s current solution in comparison to ours. The measurement method can also be used to compare the status quo to our proposal. One of the aims with the thesis project is therefore to design an objective measurement tool, which is applicable to the platform software at Scania. This tool will measure the software quality metrics coupling and cohesion, where our goal is to propose a software architectural design with looser coupling between the software components and higher cohesion within each software component.

To verify the architectural design of the platform software, which is required by ISO 26262, the design needs to reflect the software safety requirements as well as the requirements of the software components. Another aim with the thesis project is therefore to formulate these requirements in a structured way which enables this.

1.4 Delimitations

The thesis was delimited to only consider a small part of the platform software’s architectural design. The main focus is on the sensor drivers and the real-time database. The sensor drivers are performing the input signals’ conversion from the output of the A/D-converter to their engineering units during run-time, and the real-time database contains signals that are exchanged between different software
components in the ECU. Only the basic functionality of the real-time database has been implemented in source code in this thesis.

The ECU that the thesis has been focusing on is the 3rd generation Exhaust Emission Control system (EEC3), which handles the exhaust emission of the vehicle. The scheduling processes of the platform software have not been considered. Only the selected parts in subsections 2.3.2 and 2.3.3 regarding the software architectural design and the software unit design have been considered in ISO 26262. The focus is on the requirements regarding the complexity and the verification of the software architectural design.

1.5 Methodology

This section describes the methodology chosen for the thesis project. The methodology used during the research process is presented in subsection 1.5.1 and the software development methodology is presented in subsection 1.5.2.

1.5.1 Research Methodology

The background study for the thesis project was done by reading books, articles and related Master’s thesis reports at Scania. A careful reading of ISO 26262 was also done in order to gain knowledge about the structure of the standard as well as the requirements of the standard. In order to gain an even better understanding of the meaning of ISO 26262’s requirements, we have attended weekly study circles during the first month of the project. The study circles were held together with other thesis workers, our industrial supervisor Mattias Nyberg and the industrial researcher Jonas Westman, who are both specialized in ISO 26262.

We have also conducted interviews with employees at Scania in order to gain a better understanding of the thoughts behind the architectural design decisions made for the platform software, particularly the real-time database.

1.5.2 Software Development Methodology

The work carried out in the thesis project was done by following an agile software development method, so that it would be in accordance to the way the research-and-development department at Scania works today. The agile method is based on multiple iterations where requirements and functionalities are incremented as the project proceeds. In this way, useful and working software is developed as early as possible while it is responsive to sudden changes to the requirements. In contrast to the waterfall model, no long-term planning is done since software development is not considered to be predictable enough [26].

A frugal engineering approach has been taken in order to break up the complex structure of Scania’s current implementation, and re-building each software component in a more economical manner. For the development of the architectural design of our solutions, we have iterated through the following steps three times:
1. Understand the current design and implementation of the platform software in an ECU.

2. Understand current and future requirements and needs from both Scania and ISO 26262.

3. Evaluate the current software architecture.

4. Suggest a new software architectural design for the platform software.

5. Implement the suggested architectural design.

In the first iteration, only the sensor drivers in the platform software were considered where some of the requirements were left out, such as performing diagnostics on the signals. In the second iteration, these requirements were included and diagnostics were performed on the signals. Different concepts of the real-time database in the platform software were developed and evaluated during the second iteration.

The third and last iteration was considering the verifiability of the software architectural design, as well as verifying and validating the functionality of our developed software. The time frame for the first and second iteration was one month each, while the third iteration lasted for two months.

The measurement tool was developed as two separate modules; one that extracts the necessary information from the source code, and one that implements the chosen algorithm for computing coupling and cohesion. This was done in order to increase the efficiency, since the first part of the tool was dependent on the thesis work [19] by Oskar Molin, who was working in parallel with us.

1.6 Summary of the Results and Conclusions

The thesis project resulted in three alternative solutions for the software architectural design of the sensor drivers, and four alternative concepts for the real-time database in the platform software. All our solutions for the sensor drivers have lower complexity, better performance and better compliance with ISO 26262 than Scania’s current solution. A measurement tool for measuring the software quality metrics coupling and cohesion has been developed in the thesis work and the software architectural design was verified using a structured framework. The main conclusion from this thesis is that there is great potential for Scania to reduce the complexity of the software architecture and comply with ISO 26262.
2. Frame-of-Reference

This chapter presents Scania’s architectural design of the platform software in one of their ECUs. Different concepts for fault-tolerant control systems are described in this chapter, as well as the requirements from ISO 26262 and the methods used to verify the compliance with ISO 26262.

2.1 Scania’s Architectural Design of the Platform Software

It is important for a complex structure, such as the platform software in Scania’s ECUs, to be built on a solid foundation \[15\]. To ensure that the software is stable, the structure needs to meet all technical and operational requirements of the system. This can be obtained by a software architecture, which denotes the high level structures of a software system \[2\]. One of the aims with the design of a software architecture is to expose the structure of the system without showing the implementation details \[15\].

The architectural design of the platform software in Scania’s ECUs encompasses design decisions about the organization of the software elements and their relations to each other. It is developed in-house by Scania and has functionality similar to a real-time operating system, which makes use of the cyclic executive \[17\] scheduling algorithm.

Scania has developed different architectural designs of the platform software for different ECUs, due to different requirements for the ECUs. This thesis is focusing on EEC3, which is described in subsection 2.1.1. The software architectural design of EEC3 is described in subsections 2.1.2 and 2.1.3. Only a small part of the platform software’s architectural design has been considered in this thesis, namely the sensor drivers and the real-time database. The definitions and descriptions of these are presented in subsections 2.1.4 and 2.1.5.

2.1.1 Description of the 3rd Generation Exhaust Emission Control System

EEC3 is the ECU with the responsibility for controlling the exhaust and aftertreatment systems in many of Scania’s vehicles. One of the aftertreatment systems is the Selective Catalytic Reduction (SCR) system. The SCR-system’s main task is to convert the harmful nitrogen oxide (NOx) gases produced by the engine’s combustion into diatomic nitrogen and water. This is done by making the exhaust gases react with ammonia which gives carbon dioxide as a reaction product. The ammonia is supplied to the catalyst in the form of a 32.5 % urea and water-solution called AdBlue \[10\].
Another aftertreatment system is the Diesel Particulate Filter (DPF). This filter is designed to remove the diesel particulate matter or soot from the exhaust gas of a diesel engine and can do so very efficiently. However, in order for function properly the DPF needs to be regenerated (i.e. cleaned) on a regular basis. One way of achieving this regeneration is by supplying diesel fuel to the exhaust upstream the oxidation catalyst [10].

EEC3 manages signals from many sensors such as NOx-sensors, temperature sensors, and pressure sensors. The signals are used for performing calculations and diagnostics of the system and will determine the output signals, which control actuators such as pumps and injectors. These pumps and injectors are used for supplying AdBlue to the SCR-system and diesel fuel to the DPF-system.

2.1.2 The Architectural Design of the Platform Software in EEC3

The architectural design of the platform software in EEC3 encompasses design decisions for all software in the system. In EEC3 the platform software’s architectural design has a layered structure with different levels of abstraction. The layered structure consists of eight different layers and is described in [4]. The layered structure is also reflected in the folder structure of the source code. The overall goal is to limit the number of relations and keep the dependencies between the layers one-directional as much as possible [4].

Each layer consists of several modules and/or managers that are implemented in the procedural programming language C. A manager is a software component, which according to [4] is defined as a logical group of modules that share a common functionality and can have much communication between each other. A module is also a software component, which is defined as a c-file with its associated header-file and calibration-file [4]. The c-file contains functions that perform a specified task, the header-file defines the interface to the module’s environment, and the calibration-file defines any constants that are used by the functions in the c-file.

2.1.3 Description of the Layers in the Software Architectural Design of EEC3

The application layer (APPL) contains modules that control the hydraulics and incorporates the diagnostics of both the SCR-system and the DPF-system. These systems may exist separately or together. The virtual sensor layer (VSEN) has modules that are responsible for filtering and diagnosing sensor signals. Since some signals are purely calculated from models, the modules in the VSEN-layer also have the responsibility for determining which of the incoming signals that should be used by the application, and which that should come from the modeled signals.

The modules in the low level application layer (LLAP) are handling all of the input- and output (I/O) signals of the ECU. This includes all sensors, actuators and CAN-messages that are used for communicating with other ECUs in the vehicle. This layer is connected to the physical I/O-ports via the common platform (COMP).
COMP is a standard package that is shared by all of the ECUs in the vehicle where one of its purposes is to abstract the hardware details to the applications in the ECU. It contains reusable software functions that constitute the interface to the hardware, as well as software that handles the ECUs’ fault-tolerance mechanisms.

The system manager (SYSM) consists of modules that are responsible for the initiation and run-time scheduling of all modules, including the fault-tolerance mechanism in COMP. The main part of the scheduling mechanism consists of time based loops, which are executed on a regular basis. The file handler layer (FILE) has modules that handle file operations and the controlling of the memory systems in the ECU.

The real-time database (RTDB) contains signals that are exchanged between different managers or modules. The signals stored in this layer are typically changed over time. It provides functionality for writing and reading the data as well as overwriting the signals for testing- and aftermarket purposes. Finally, the utilities layer (UTLS) consists of modules that contain reusable software functions. Examples of functions residing in the UTLS-layer are filtering-, interpolation- and other mathematical-functions.

2.1.4 The Sensor Drivers in EEC3

In the LLAP-layer, the sensor drivers are performing the input signals’ conversion from the output of the A/D-converter to their engineering units during run-time. To demonstrate the current software architectural design of the sensor drivers in EEC3, we let the pressure sensor driver serve as an illustrative example. The pressure sensor driver’s main task is to provide the pressure value together with a signal status to the software applications for them to use. The signal status is reflecting the quality and origin of the pressure value. The pressure sensor connected to EEC3 is used for measuring the pressure of AdBlue fluid for the SCR-system or the pressure of diesel fuel for the DPF-system.

Figure 2.1 illustrates the basic workflow of the process. It starts with the A/D-converter, which samples the analog sensor signal and converts it to a digital value. This is done at a specific frequency and the samples are stored in a buffer register containing the most recent samples. The pressure sensor driver reads the sample data from the buffer register and converts it to mV.

In order to ensure a steady and accurate pressure value, the signal is low-pass filtered before it is converted to the engineering unit mBar. Furthermore, the voltage value is diagnosed to check whether a fault has occurred or not. Lastly, the voltage value will be translated from mV to mBar according to the data sheet for the specific sensor. When the most recent pressure value has been calculated and diagnosed by the pressure sensor driver, it is passed to the device manager together with its associated signal status. The device manager, which also resides in the LLAP-layer and handles all of the I/O-units in EEC3, makes the pressure value available for the applications to use by writing it to the real-time database.
2.1.5 The Real-Time Database in EEC3

A real-time database differs from traditional databases since it handles data that is time-constrained [15]. The purpose of the real-time database in Scania’s ECUs is to gather all signals that should be exchanged between managers or modules in all of the different software layers [4]. Typically it is volatile data that are changed over time. One of the requirements of the real-time database is to provide functionality for the four variable actions: read, write, force and release. Providing functionality of the four variable actions is the only requirement considered in this thesis.

When the real-time database performs the variable actions read and write, it should simply read and write a new value to the variable in the database. The variable action force is used for overriding the data in the variable. When the force-action is used, the variable shall be overwritten with a new value and any attempts of performing the write-action should silently fail. This is useful for both testing purposes and the aftermarket. The variable action release is used for releasing a variable from the forced-state so that the write-action can write a new value to the variable as normal.

2.2 Fault-Tolerant Control of Electrical Control Units

Fault-tolerant control methods have been developed with the capability of detecting fault occurrences and retaining satisfactory system performance in the presence of faults [9]. The main purpose of the fault-tolerant control in the ECUs developed by Scania is to increase the uptime of the vehicle, the safety of the vehicle, enhance repair and fulfill On-Board Diagnosis (OBD) regulations [3]. This thesis discusses two different approaches for the architecture of fault-tolerant control; a centralized architecture and a decentralized architecture. The centralized architecture, which is the architecture Scania has implemented in most of their ECUs, is described in subsection 2.2.1 and the decentralized architecture is described in subsection 2.2.2.

2.2.1 Scania’s Centralized Architecture for Fault-Tolerant Control

A diagnostic framework has been developed by Scania and is denoted DIMA platform. The DIMA platform consists of a basic software in COMP denoted DIMA-BSW, a PC-tool denoted DIMA-tool, and the architectural specification document TB4093 [3]. DIMA-BSW is designed as a centralized diagnostic manager which is responsible for all fault-tolerant control and system degradations in the ECUs.
The system degradations are decided by DIMA-BSW dependent on the test results from the subsystems in the ECU. Different degradation actions will be taken depending on which parts of the system a fault has been detected in. In order for DIMA-BSW to decide which degradation action to take, information about the fault is stored in the form of diagnostic trouble codes (DTCs). All of the software modules’ DTC information in the ECU is stored in one single module. In EEC3 this module consists of more than thousands of lines of code and is edited by the DIMA-tool. It contains references to the diagnostic tests and their DTCs as well as information about the connected physical components.

Each signal from a subsystem in EEC3 has a signal status associated with it. These signal statuses report the quality and origin of the signal (i.e. if a fault is pending or if the signal is a replacement signal). The signal statuses also contain information about the component status, which is communicated to the subsystems through DIMA-BSW.

2.2.2 A Decentralized Service Based Architecture for Fault-Tolerant Control Systems

In [23] the concept of a decentralized service based architecture for fault-tolerant control is introduced. By distributing the fault-tolerant control to the software modules, several benefits can be obtained.

First of all, the system will have a faster response time for the detected faults since the fault-tolerant control algorithms are executed locally in the software modules. In this way the response actions will not be delayed by the communication network for the centralized diagnostic manager. A faster response time will increase the safety of the vehicle since it can guarantee a faster response action when a fault has been detected.

Another benefit is that the complexity of the diagnostic framework will be significantly reduced. This will reduce the risk of failures resulting from high complexity as well as making the software easier to understand for software developers. One more benefit with a decentralized architecture for fault-tolerant control is that it will be easier to maintain for software developers if the responsibility for the fault handling is distributed throughout the system, instead of having everything kept in the same module.

In the concept of a service based architecture [23], both the hardware- and software subsystems are considered as service providers. Each service provider would then provide a service and a service status to a customer, which itself is a service provider. In [23] it is suggested to only have three classes of statuses, which are denoted nominal (NOM), disturbed (DIST) and unavailable (UNA). The service will have the status NOM when no fault has been detected. The statuses DIST and UNA may or may not be needed depending on the specific case. DIST means that the service provider only can provide a degraded service, while UNA means that the service from the service provider is unavailable. The DIST-status can also be extended to several levels of disturbed such as DIST1, DIST2 and DIST3 etc.
2.3 Requirements for Product Development at the Software Level from ISO 26262

ISO 26262 [11] is an automotive specific adaption of the IEC 61508 functional safety standard for electrical and electronic (E/E) systems. Like IEC 61508, ISO 26262 is a risk-based safety standard where hazardous situations are assessed and safety measures are defined to avoid or control systematic failures, detect or control random hardware failures, or mitigate their effects. In general ISO 26262 [11]:

- Provides an automotive safety lifecycle (management, development, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases.
- Provides an automotive specific risk-based approach for determining Automotive Safety Integrity Levels (ASILs), which are defined in subsection 2.3.1.
- Uses ASILs for specifying the item’s necessary safety requirements for achieving an acceptable residual risk.
- Provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved.

The standard is divided into ten volumes where volume one is a vocabulary, volume two to nine covers the automotive safety lifecycle and volume ten is a guideline for using the standard itself. The volume of interest in this thesis is volume six, which covers the product development at the software level, especially the software architectural design and the software unit design and implementation. These sections are described in subsections 2.3.2 and 2.3.3 respectively.

For the software architectural design, ISO 26262 states that it is important to have restricted coupling between software components and high cohesion within each software component. The definition of the software quality metrics coupling and cohesion, which were invented by [6], is described in subsection 2.3.4 and different methods of measuring these metrics are presented in subsection 2.3.5. The algorithm for the method of choice in the thesis is described in subsection 2.3.6.

ISO 26262 also requires that the software architectural design shall be verifiable. This can be achieved in a structured way by applying the method described in 2.3.7.

2.3.1 Automotive Safety Integrity Levels

An ASIL shall be determined for each hazardous event using the parameters severity, probability of exposure and controllability. ISO 26262 has defined four classes of ASILs: ASIL A, ASIL B, ASIL C and ASIL D, where ASIL A is the lowest safety integrity level and ASIL D is the highest level. In addition to these four ASILs, the class QM (quality management) denotes that no requirement is needed
for complying with ISO 26262. For each method in ISO 26262, the degree of recommendation to use the corresponding method depends on the ASIL and is categorized as follows [11]:

- ‘++’ indicates that the method is highly recommended for the identified ASIL;
- ‘+’ indicates that the method is recommended for the identified ASIL;
- ‘◦’ indicates that the method has no recommendation for or against its usage for the identified ASIL.

In this thesis the aim is to fulfill the requirements that are highly recommended by ISO 26262 for complying with ASIL C.

### 2.3.2 Requirements for the Software Architectural Design from ISO 26262

The requirements for the software architectural design are treated in chapter 7 in the sixth volume of ISO 26262. The objectives with this chapter are to assure that the software architectural design realizes the software safety requirements and to verify the software architectural design. According to ISO 26262 [11] the software architectural design represents all software components and their interactions in a hierarchical structure 1. Here a software component is defined as one or more software units. A software unit is in turn defined as an atomic level software component of the software architecture that can be subjected to standalone testing, and maps to a module or a manager in Scania’s terminology. The intention of the specification of the software architectural design is to ensure that it provides the means to implement the software safety requirements and to manage the complexity of the software development.

In the scope of this thesis only a subset of the requirement- and recommendation clauses in chapter 7 of ISO 26262:6 is considered, namely those that reflect the complexity and the verification of the software architectural design. The subset considered in this thesis consists of the following requirement- and recommendation clauses:

**(7.4.2-a)** During the development of the software architectural design, consider the verifiability of the software architectural design. This implies bi-directional traceability between the software architectural design and the software safety requirements.

**(7.4.3)** In order to avoid failures resulting from high complexity, the software architectural design shall exhibit the properties modularity, encapsulation and simplicity by the use of the principles listed in table 2.1.

---

1In an hierarchical structure, all subentities of a larger entity must be identifiable to be contained within the larger entity [31].
Table 2.1: Principles for software architectural design \[11\].

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Hierarchical structure of software components</td>
<td>++</td>
</tr>
<tr>
<td>1b Restricted size of software components(^a)</td>
<td>++</td>
</tr>
<tr>
<td>1c Restricted size of interfaces(^b)</td>
<td>+</td>
</tr>
<tr>
<td>1d High cohesion within each software component(^c)</td>
<td>+</td>
</tr>
<tr>
<td>1e Restricted coupling between software components(^d), (^b), (^c)</td>
<td>+</td>
</tr>
<tr>
<td>1f Appropriate scheduling properties</td>
<td>++</td>
</tr>
<tr>
<td>1g Restricted use of interrupts(^a), (^d)</td>
<td>+</td>
</tr>
</tbody>
</table>

\(^a\) In methods 1b, 1c, 1e and 1g “restricted” means to minimize in balance with other design considerations.

\(^b\) Methods 1d and 1e can, for example, be achieved by separation of concerns which refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose.

\(^c\) Method 1e addresses the limitation of the external coupling of software components.

\(^d\) Any interrupts used have to be priority-based.

(7.4.4) The software architectural design shall be developed down to the level where all software units are identified.

(7.4.9) The software safety requirements shall be allocated to the software components. As a result, each software component shall be developed in compliance with the highest ASIL of any of the requirements allocated to it.

2.3.3 Requirements for the Software Unit Design and Implementation from ISO 26262

The requirements of the software unit design and implementation are treated in chapter 8 of ISO 26262:6. The objectives with this chapter are to specify the design of the software units in accordance with the software architectural design and the associated software safety requirements.

In the scope of this thesis only a subset of the requirement- and recommendation clauses in chapter 8 of ISO 26262:6 is considered. The subset contains only the requirements that are considered to affect the complexity of the platform software and consists of the following requirement- and recommendation clauses:

(8.4.4-d-h) Design principles for software unit design and implementation at the source code level as listed in table 2.2 shall be applied to achieve the principles simplicity, readability, robustness, suitability for software modification and testability.
Table 2.2: Design principles for software unit design and implementation [11].

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a One entry and one exit point in subprograms and functions&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1b No dynamic objects or variables, or else online test during their creation&lt;sup&gt;a,b&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1c Initialization of variables</td>
<td>++</td>
</tr>
<tr>
<td>1d No multiple use of variable names&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1e Avoid global variables or else justify their usage&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1f Limited use of pointers&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1g No implicit type conversions&lt;sup&gt;a,b&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1h No hidden data flow or control flow&lt;sup&gt;c&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1i No unconditional jumps&lt;sup&gt;a,b,c&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1j No recursions</td>
<td>++</td>
</tr>
</tbody>
</table>

<sup>a</sup> Methods 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

<sup>b</sup> Methods 1g and 1i are not applicable in assembler programming.

<sup>c</sup> Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

2.3.4 Definition of Coupling and Cohesion

In [32], coupling is defined as the degree of interdependence between software modules. For example, consider two software modules $a$ and $b$. Then coupling quantifies how much information you need from $b$ in order to make a change in $a$. If modules $a$ and $b$ are designed in such way that $a$ and $b$ are completely independent of each other, then there would not be any coupling between them by definition.

Cohesion is defined in [32] as a measure that quantifies how tight or related a module’s internal elements are to one another. It is also described as the "glue factor" within a module [7]. Cohesion is usually normalized to a value between zero and one, where zero represents no relation between the module’s internal elements and one represents a complete relation between the module’s internal elements where every element is related to each other.

Even though these two software quality metrics do not necessary reflect the complexity of the implementation, they reflect the complexity of the architectural design. It is desirable to design a software architecture with loose coupling between the software components and high cohesion within each software component, since it leads to reduced maintenance costs [6]. There are different types of coupling and cohesion that can occur [7], and they are presented in tables 2.3 and 2.4 on page 14. The different types of coupling in table 2.3 are ranked in the order from loose coupling to tight coupling, and the different types of cohesion in table 2.4 are ranked in the order from high cohesion to low cohesion.
Table 2.3: Different kinds of coupling that can occur according to [7]. These are ranked in the order from loose coupling to tight coupling, where data coupling by value is ranked as the loosest one.

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurs through</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data coupling by value</td>
<td>Data transfer</td>
</tr>
<tr>
<td>Data coupling by reference</td>
<td>Pointer transfer</td>
</tr>
<tr>
<td>Stamp coupling</td>
<td>Sharing of a global struct via pointer transfer</td>
</tr>
<tr>
<td>Common coupling</td>
<td>Global variables</td>
</tr>
<tr>
<td>Content coupling</td>
<td>Code sharing</td>
</tr>
</tbody>
</table>

Table 2.4: Different kinds of cohesion that can occur according to [7]. These are ranked in the order from high cohesion to low cohesion, where functional cohesion is ranked as the highest one.

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurs when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional cohesion</td>
<td>Each functional module performs a single function</td>
</tr>
<tr>
<td>Sequential cohesion</td>
<td>The elements are organized as a sequence of operations</td>
</tr>
<tr>
<td>Communicational cohesion</td>
<td>The elements of a module act on common data</td>
</tr>
<tr>
<td>Procedural cohesion</td>
<td>Each element is part of a well-defined procedure</td>
</tr>
<tr>
<td>Temporal cohesion</td>
<td>The elements of a module are related by time</td>
</tr>
<tr>
<td>Logical cohesion</td>
<td>The elements of a module are grouped together only because they are logically connected</td>
</tr>
<tr>
<td>Coincidental cohesion</td>
<td>The elements of a module are unrelated</td>
</tr>
</tbody>
</table>

2.3.5 Methods for Measuring Coupling and Cohesion

There are several methods that try to quantify the attributes in tables 2.3 and 2.4 or a subset of them. To measure cohesion, [7], suggests two different approaches; from a problem view or from a program view. Depending on which approach is taken, the set of possible ways for cohesion to occur will be a subset of all possible ways [7]. In this thesis we have chosen to approach it from a program view since we are measuring source code that has already been implemented. With this view, the functional and sequential cohesion are disregarded.
Though many of the methods focus on object-oriented designs of software, they are not applicable here since Scania has implemented their source code in the procedural language C. By using the information theory approach introduced in [1], we can measure coupling and cohesion, regardless if the source code is implemented in an object-oriented or procedural programming language. The algorithm for this method is described in subsection 2.3.6 and works on every abstraction of the source code that results in a graph. The key concept in [1] compared to classical measurements such as [8], is that [1] models the graphical abstraction of the source code as a probability distribution of patterns, while classical methods base their measurements on counts of different attributes.

A pattern for node \( n \) in the graphical abstraction of the source code is considered to be its corresponding tuple \( E_n = \{e_1, e_2, \ldots, e_m\} \), where each element in the tuple represents edges to other nodes that are directly connected to \( n \). Each element in the tuple can adopt the values one or zero depending on whether \( n \) is connected to edge \( e_i \) or not, where \( i \in [1, m] \) and \( m \) is the total number of edges. By calculating the probability distribution of the patterns, adoption of information theory is possible. We adopt the idea from [1] that the graphical abstraction of the source code embodies design decisions as information.

### 2.3.6 Algorithm for Measuring Coupling and Cohesion

We will demonstrate the measurement algorithm presented in [1] using the example graph \( G \) in figure 2.2. The example graph \( G \) consists of two modules with four nodes and three edges.

![Figure 2.2: Example graph G with the modules m₁ and m₂ containing four nodes and three edges.](image)

**Algorithm for measuring coupling**

The coupling of \( G \) is calculated from a subgraph \( S \), which is illustrated in figure 2.3a. The subgraph \( S \) consists of all nodes and intermodule edges\(^2\) in \( G \), plus one additional environment node. The environment node is representing the fact that interfaces to the system’s environment are not of interest in the calculations.

\(^2\)An intermodule edge has end-points in two different modules.
The properties of $S$ is captured in the node × edge table in figure 2.3b. For each row in the table, the probability distribution $p_{L(i)}$ of the row patterns is calculated from the proportion of distinct patterns.

![Subgraph S and its representation as a node × edge table.](image)

Figure 2.3: The subgraph $S$ and its representation as a node × edge table. The probability distribution $p_{L(i)}$ represents the proportions of distinct row patterns.

From subgraph $S$, the entropy $H(S)$ for the distribution of row patterns is calculated using equation (2.1).

$$H(S) = \sum_{i=0}^{n} \frac{1}{n+1} \left(- \log_2 p_{L(i)} \right)$$

(2.1)

Since entropy represents the average information per node, $H(S)$ is multiplied by the total number of nodes in order to obtain the total amount of information, $I(S)$, in the structure of the graph. This is done by the use of equation (2.2).

$$I(S) = H(S)(n + 1) = \sum_{i=0}^{n} \left(- \log_2 p_{L(i)} \right)$$

(2.2)

Furthermore, information theory states that entropy of a joint distribution is less than or equal to the sum of the entropy of its components. Consider each subgraph $S_i$ to be a component of the joint distribution $S$, where $S_i$ contains all nodes in $S$ and all edges that has node $i$ as end-point, then:

$$\sum_{i=0}^{n} H(S_i) \geq H(S)$$

(2.3)

The difference in this inequality is shown by [29] to be a measure of the relationships among the components $C(S)$ according to equation (2.4).

$$C(S) = \sum_{i=0}^{n} H(S_i) - H(S)$$

(2.4)
By multiplying \( C(S) \) with the total number of nodes, an estimate of the minimum description length of the relationships is given in equation (2.5), and is considered to represent the coupling of the graph \( G \)

\[
(n + 1)C(s) = \sum_{i=1}^{n} I(S_i) - I(S) = Coupling(G) \tag{2.5}
\]

**Algorithm for measuring cohesion**

The cohesion of \( G \) is calculated from the subgraph \( S' \), consisting of all nodes and intramodule edges\(^3\) in \( G \), plus one additional environment node. The environment node is representing the fact that interfaces to the system’s environment are not of interest in these calculations as well. The subgraph \( S' \) is illustrated in figure 2.4a and the properties of \( S' \) is captured in the node × edge table in figure 2.4b including the probability distribution \( p_{L(i)} \) of the row patterns.

![Figure 2.4](image)

Figure 2.4: The subgraph \( S' \) and its representation as a node × edge table. The probability distribution \( p_{L(i)} \) represents the proportions of distinct row patterns.

Substituting \( S \) with \( S' \) in equations (2.1) and (2.2), as well as substituting \( S_i \) with \( S'_i \) in equations (2.3) and (2.4), intramodule coupling is considered to be represented by equation (2.6).

\[
\sum_{i=1}^{n} I(S'_i) - I(S') \tag{2.6}
\]

Cohesion is by definition a normalized value in the interval \([0, 1]\). In order to normalize intramodule coupling to represent cohesion, consider the graph \( G' \) containing all nodes in \( G \) and a set of edges that connects all nodes to every other node in the module (the most cohesive graph possible). Then the cohesion of graph \( G \) is given by equation (2.7), when equations (2.1)-(2.4) are applied to \( G' \).

\[
Cohesion(G) = \frac{\text{intramoduleCoupling}(G)}{\text{intramoduleCoupling}(G')} \tag{2.7}
\]

\(^3\)An intramodule edge has end-points in the same module.
2.3.7 Verification of the Software Architectural Design

Verification of a software architectural design can be done in many different ways depending on the objective of the verification. In this case, the objective is to verify the compliance with the functional software safety requirements of the system in order to comply with ISO 26262.

For the verification of the software architectural design, the architectural structure is represented according to the framework given in section Components and Systems in [22]. With this representation, the verification at an atomic level can be approached by structuring the software safety requirements according to [30] using contract theory. This will enable verification of the consistency of the functional software safety requirements in accordance with ISO 26262.

A complete design of the architecture from an atomic leveled view consists of a set of relations between software units. The software units are considered to consist of a set of variables called ports and the relations between software units are represented as connections between ports [22]. In this thesis the relations between software units are considered to be data flow.

All existing software units in a subsystem are clustered together into one artificial functional component. This enables the possibility of adding a set of environment ports to the system, which is necessary in order to specify a contract for the actual functionality provided by the subsystem.

A contract $C$ is modeled for each software unit. The contract $C$ contains a set of requirements $R$ that are to be satisfied under the assumptions $A$ with the behavior $B$. An implementation $M$ satisfies $C$ if equation (2.8) holds [30]. This equation is graphically illustrated in the venn diagram\(^4\) in figure 2.5.

$$A \cap B \subseteq R \quad (2.8)$$

![Figure 2.5: Equation (2.8) is illustrated in this venn-diagram.](image)

Furthermore, we will consider all individual software units from an environmental-centric point of view. In an environmental-centric point of view, each software unit has knowledge about the environment and all ports on the same hierarchical level.

\(^4\)A venn diagram shows the logical relation between sets.
We will demonstrate the structure of formulating software safety requirements using an example software unit temp, where a block diagram [16] with ports as interfaces is illustrated in figure 2.6a. The block diagram illustrates how temp is related to its environment, where the relations are undirected. The software unit temp has two ports, \( p_1 \) and \( p_2 \), and the real temperature value is modeled as port \( p_3 \).

In this example the requirement for temp, \( R_{1:1} \), is to provide \( p_1 = p_3 \). In order to satisfy the correct behavior for temp, assumption \( A_{1:1} \) is made where \( p_2 = f(p_3) \). With this assumption and requirement, equation (2.8) implies that the behavior of temp shall be \( p_1 = f^{-1}(p_2) \), which yields \( p_1 = p_3 \). The contract of temp, \( C_{\text{temp}} \), consists of requirement \( R_{1:1} \) and assumption \( A_{1:1} \) and is presented in figure 2.6b.

![Figure 2.6: Example of a software unit temp with its associated contract \( C_{\text{temp}} \). The block diagram for temp is illustrated in (a) and the contract \( C_{\text{temp}} \) is illustrated in (b).](image)

The relationship between requirement \( R_{1:1} \) and assumption \( A_{1:1} \) can be illustrated in the directed graph in figure 2.7. The directed graph is useful when verifying the consistency of the requirements, especially when an assumption is depending on several requirements. In this case the directed graph is very trivial since requirement \( R_{1:1} \) is only dependent on assumption \( A_{1:1} \), which is fulfilled by the environment.

![Figure 2.7: Directed graph for verifying the consistency of the software safety requirements for the software architectural design of temp.](image)
3. Analysis

In this chapter, a gap analysis between the requirements from ISO 26262 and Scania’s current implementation of the sensor drivers and the real-time database in EEC3 is presented. Both the software architectural design and the design of the software units residing in the architecture are analyzed.

3.1 Gap Analysis between ISO 26262 and Scania’s Current Software Architectural Design in EEC3

A gap analysis between ISO 26262 and Scania’s current software architectural design in EEC3 is presented in this section. The analysis is based on the gap analysis from [31]. Though the analysis from [31] focuses on the overall gap between the software development at Scania and the requirements from ISO 26262, this analysis focuses only on the sensor drivers and the real-time database in EEC3.

The analysis is presented in this section where each requirement is categorized as fulfilled, partially fulfilled or not fulfilled. Only the requirements regarding complexity and verification of the software architecture are considered, and Scania’s aim is to only fulfill the requirements that are highly recommended for ASIL C.

NOTE TO THE READER: The analysis in this section has been excluded due to the nature of the content.

3.2 Gap Analysis between ISO 26262 and Scania’s Current Software Unit Design in EEC3

The software unit design of the software components residing in the sensor drivers and the real-time database in EEC3 have been analyzed in order to evaluate their compliance with the requirements from ISO 26262. The analysis presented in this section is also based on the gap analysis from [31]. Only the requirements regarding complexity and verification of the software architecture are considered, and Scania’s aim is to only fulfill the requirements that are highly recommended for ASIL C. Many of the requirements are covered by "C-Programming rules for embedded software in vehicle systems" [20], which have been followed by the software developers at Scania. Most of the rules in [20] are identical to rules in MISRA-C:2004 [21].

NOTE TO THE READER: The analysis in this section has been excluded due to the nature of the content.
4. Software Architectural Design

In this chapter we present alternative concepts for the software architectural design of the pressure sensor driver and the real-time database. These have been specifically developed in order to reduce the complexity of the architecture and fulfill the requirements imposed by ISO 26262.

4.1 Design and Implementation of the Pressure Sensor Driver in EEC3

Based on the outcome of the analysis in chapter 3, there are requirements from ISO 26262 that could be improved in Scania’s current software architectural design of the sensor drivers in EEC3. For the pressure sensor driver, which is described in 2.1.4, the requirements that could be improved are: "restricted coupling between software components", "high cohesion within each software component", "limited use of pointers" and "no hidden data flow".

This section presents the concepts of three alternative solutions for the architectural design of the pressure sensor driver. These have been specifically developed in order to reduce the complexity of the architecture and fulfill the requirements imposed by ISO 26262. The concepts are denoted Pres v1, Pres v2 and Pres v3, and are described in subsections 4.1.1, 4.1.2 and 4.1.3 respectively. All of these concepts are scalable to the rest of the sensor drivers in EEC3.

The main idea for decreasing the complexity is to modify the internal structure of the sensor driver and avoid the reusable software functions that reside in COMP and the UTLS-layer. This has been done in various degrees in our three different concepts, so that the trade-offs between the different modifications can be evaluated.

The solutions have been developed using the code refactoring technique. All of our solutions implement the concept of a decentralized service based architecture for fault-tolerant control, which is presented in subsection 4.1.4. The solutions have been fully implemented in source code and tested using a virtual testbed, which is described in subsection 4.1.5.

4.1.1 Pres v1 - First Concept for the Software Architectural Design of the Pressure Sensor Driver in EEC3

Pres v1, which is the first concept we developed for the pressure sensor driver in EEC3, is based on Scania’s current architectural design but has major differences in both the internal and the modular structure of the sensor driver. The internal

---

1 Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" [12].
The internal and modular structure of *Pres v1*, illustrated in (a) and (b) respectively. The arrows in (a) represent the direction of the internal data flow, where dashed arrows represent hidden data flow. The colors of the modules in (b) represent which software architectural layer they belong to.

Structure of the pressure sensor driver called *pres_v1* is illustrated in figure 4.1a. Except for the initialization-function and the diagnostics-function, everything is handled in one single function.

The initialization-function connects the sensor driver to its associated channel in the A/D-converter the same way Scania’s current implementation does. The concept of having a fundamental struct with pointers to global variables at file scope is abandoned and the variables are stored locally within the functions instead. This decreases the number of pointers and reduces the hidden data flow. The device manager retrieves the pressure value with its signal status from the sensor driver’s main-function.

One of the major differences between *Pres v1* and Scania’s current solution is the dependencies to other modules. In this concept the functionality of the reusable software functions in COMP is done in module *pres_v1* instead. The reading of the samples from the A/D-converter is done in *pres_v1* as well as the conversion to mV. Some of the features with avoiding the software functions in COMP are looser coupling, fewer code lines, faster execution time and less memory allocation on the stack. The total amount of code lines will however be increased if this concept is scaled to all of the sensor drivers in EEC3. The work flow of *pres_v1* and the use of utility-functions in the UTLS-layer are the same as Scania’s current implementation. Figure 4.1b illustrates the modular structure of *Pres v1* and the pseudo code for *Pres v1* is presented in algorithm 1. The pseudo code for the diagnostics-function is described in subsection 4.1.4.
Algorithm 1 Pseudo code for Pres v1

1: function MAIN FUNCTION(pressure sensor)
2:     for each sample in ADC buffer do
3:         check origin of signal value
4:         convert sample data to mV
5:         filter signal using UTLS-function
6:     end for
7:     recognize if signal comes from AdBlue-sensor or DPF fuel-sensor
8:     status = DIAGNOSTICS(filtered signal)
9:     if status is NOM then
10:         calculate pressure using UTLS-function
11:     else if status is DIST then
12:         freeze pressure value
13:     else if status is UNA then
14:         set pressure to a predefined pressure value
15:     end if
16:     return pressure and status
17: end function

4.1.2 Pres v2 - Second Concept for the Software Architectural Design of the Pressure Sensor Driver in EEC3

The second concept for the pressure sensor driver in EEC3, Pres v2, is based on Pres v1 but has changes in both the internal structure and the modular structure. The internal structure of the sensor driver pres_v1 is changed so that it is split up into two separate modules; pres_v2_abd1 and pres_v2_dpfu. Module pres_v2_abd1 handles the pressure sensor for the AdBlue fluid and pres_v2_dpfu handles the pressure sensor for the diesel fuel in the DPF-system. Splitting the sensor drivers into two modules will increase the cohesion because having them together causes logical cohesion, which is one of the lowest form of cohesion according to table 2.4 in subsection 2.3.4. Also, since Scania’s vehicles most often do not contain both an SCR-system and a DPF-system at the same time, only one of the software modules is needed to be implemented in the ECU.

The internal structure of the pressure sensor driver in Pres v2 is also changed so that different tasks are done in separate functions within the module, similarly to Scania’s current solution. This is illustrated in figure 4.2a. Some of the features with this are higher cohesion and a more distinct internal structure. The variables are passed as arguments to the different functions in order to avoid any hidden data flow.
Figure 4.2: The internal and modular structure of Pres v2, illustrated in (a) and (b) respectively. The arrows in (a) represent the direction of the internal data flow, where dashed arrows represent hidden data flow. The colors of the modules in (b) represent which software architectural layer they belong to.

The modular structure differs from Pres v1 since the functionality of the software functions in the UTLS-layer is moved to the pressure sensor driver as well. Both the low-pass filter and interpolation function are therefore implemented in pres_v2_adbl and pres_v2_dpfu. In this way the coupling between the modules becomes even more restricted and the number of code lines is reduced, since the functionality of the utility-functions can be written more specific. The only module from the UTLS-layer that is kept is util_inte since it is storing the error messages for the mechanics in the workshops. The modular structure of our second solution for the pressure sensor driver is illustrated in figure 4.2b and the pseudo code for Pres v2 is presented in algorithm 2. The pseudo code for the diagnostics-function is described in subsection 4.1.4.
Algorithm 2 Pseudo code for Pres v2

1: function MAIN FUNCTION
2: filtered signal = FILTERVOLTAGE()
3: status = DIAGNOSTICS(filtered signal)
4: pressure = PRESSURECALCULATION(filtered signal, status)
5: return pressure and status
6: end function
7:
8: function FILTERVOLTAGE
9: for each sample in ADC buffer do
10: check origin of signal value
11: convert sample data to mV
12: filtered signal = voltage signal \times (1-C) + filtered signal \times C
13: end for
14: end function
15:
16: function PRESSURECALCULATION(filtered signal, status)
17: if status is NOM then
18: pressure = k \times filtered signal + m
19: else if status is DIST then
20: freeze pressure value
21: else if status is UNA then
22: set pressure to a predefined pressure value
23: end if
24: end function

4.1.3 Pres v3 - Third Concept for the Software Architectural Design of the Pressure Sensor Driver in EEC3

Pres v3, which is our third concept for the pressure sensor driver in EEC3, was developed as a mixture of Pres v1 and Pres v2. The internal structure is identical to Pres v1 while the modular structure is identical to Pres v2. In this way we obtain the most extreme solution with regards to number of code lines and execution time. The internal structure of the pressure sensor driver in Pres v3 is illustrated in figure 4.3a and the modular structure is illustrated in figure 4.3b. The pseudo code for Pres v3 is presented in algorithm 3 and the pseudo code for the diagnostics-function is described in subsection 4.1.4.
Figure 4.3: The internal and modular structure of Pres v3, illustrated in (a) and (b) respectively. The arrows in (a) represent the direction of the internal data flow, where dashed arrows represent hidden data flow. The colors of the modules in (b) represent which software architectural layer they belong to.

Algorithm 3 Pseudo code for Pres v3

1: function MAIN FUNCTION
2: for each sample in ADC buffer do
3:    check origin of signal value
4:    convert sample data to mV
5:    filtered signal = voltage signal×(1-C) + filtered signal×C
6: end for
7: status = DIAGNOSTICS(filtered signal)
8: if status is NOM then
9:    pressure = k×filtered signal + m
10: else if status is DIST then
11:    freeze pressure value
12: else if status is UNA then
13:    set pressure to a predefined pressure value
14: end if
15: return pressure and status
16: end function
4.1.4 Implementation of a Decentralized Service Based Architecture for Fault-Tolerant Control in the Pressure Sensor Driver

The concept of a decentralized service based architecture for fault-tolerant control, which is described in subsection 2.2.2, has been implemented in all of our three alternative solutions for the pressure sensor driver in EEC3. The pressure sensor driver is considered as a service provider that provides a service to a customer, which in this case is the device manager. The pressure value is considered as the service and the service status reflects the quality of the service. The service status is adopting any of the three statuses: NOM, DIST and UNA. The entire functionality for this is implemented in the diagnostics-function in the pressure sensor driver and the pseudo code for this function is presented in algorithm 4.

Algorithm 4 Pseudo code for the diagnostics function

1: function DIAGNOSTICS(filtered signal)
2: if filtered signal exceeds threshold value then
3: faultPending = TRUE
4: end if
5: if battery voltage exceeds threshold value then
6: status = UNA
7: else
8: if validatedFault is FALSE and faultPending is TRUE then
9: status = DIST
10: if time > 500ms then
11: validatedFault = TRUE
12: status = UNA
13: end if
14: else if validatedFault is TRUE and faultPending is FALSE then
15: status = UNA
16: if time > 500ms then
17: validatedFault = FALSE
18: status = NOM
19: end if
20: end if
21: end if
22: return status
23: end function

The functionality of the diagnostics is similar to Scania’s current implementation. The only differences are that the implementation is done within the pressure sensor driver and that the statuses Flawless, Pending error and Unavailable are changed to the statuses NOM, DIST and UNA. The functionality of the decentralized diagnostics is presented in the example in figure 4.4.

29
Figure 4.4: The functionality of the diagnostics of the pressure sensor for the DPF-system is illustrated in this figure. The top graph illustrates the raw input signal from the A/D-converter and the filtered signal. The bottom graph illustrates the output pressure value and the service status.

4.1.5 Testbed for our Concepts of the Pressure Sensor Driver in EEC3

A testbed was built in order to verify that the essential functionality of our solutions for the pressure sensor driver is equal to Scania’s current solution. A testbed is a standard software platform that allows researchers to test their new technology [16]. In software development, a testbed is used when a new software module is tested apart from the system it will later be added to. A good testbed simulates the interface of the system so that the module can be tested without the need to modify the module itself.

The testbed that was built for the pressure sensor driver was built in a way which enabled all modules that were modified in EEC3 to be tested. This was done in order to ensure a good possibility for having our prototype implementations working in an actual ECU. The testbed was built using the gcc compiler and was executed on a Windows OS computer. The sample data from the A/D-converter and the internal battery supply voltage was simulated by input values from the tester. Instead of writing the pressure value with its service status to the real-time database, it was printed out to the terminal instead.
Scania’s current implementation of the pressure sensor driver was tested first so that the behavior of the low-pass filter could be observed. Then our prototype implementations were tested, where the behavior of the low-pass filter was compared with Scania’s current implementation. After ensuring that the low-pass filter behaved as expected, a fault was induced to the system by simulating sample data that exceeded the defined threshold value. This was done in accordance to the example in figure 4.4 so that the entire functionality of the diagnostics could be tested.

4.2 Prototype Concepts for the Real-Time Database in EEC3

Similarly to the software architectural design of the sensor drivers, there are requirements from ISO 2626 that could be improved in Scania’s current software architectural design of the real-time database in EEC3. The requirements that could be improved are: "restricted coupling between software components", "high cohesion within each software component", "avoid global variables or else justify their usage", "limited use of pointers" and "no hidden data flow".

This section presents the concepts of four alternative solutions for the architectural design of the real-time database in EEC3. These have been specifically designed in order to reduce the complexity of the architecture and fulfill the requirements imposed by ISO 26262. The concepts are denoted RTDB v1, RTDB v2, RTDB v3 and RTDB v4, and are described in subsections 4.2.1, 4.2.2, 4.2.3 and 4.2.4 respectively. Only the basic functionality of these concepts have been realized in source code, so that the functionality of the concepts could be tested and verified using a virtual testbed. The testbed is described in subsection 4.2.5 and the functionality of the concepts is described in subsection 2.1.5.

4.2.1 RTDB v1 - The Concept of a Real-Time Database consisting of Global Variables

In our first concept, RTDB v1, we try to reduce the complexity of the software architecture by replacing the current real-time database mechanism with global variables. No special read-, write-, force- or release-functions are needed since all modules in the system have direct access to the variables. Some of the features with RTDB v1 are fewer code lines, faster execution time and no memory allocation on the stack. The direction of the data flow in this concept is illustrated in the example scenario in figure 4.5.
Figure 4.5: An illustration of RTDB v1, which is a real-time database consisting of global variables. In this example, there are two applications reading from three managers using global variables, where the variables are of two different variable types. The arrows represent the direction of the data flow, where dashed arrows represent hidden data flow.

Each variable in RTDB v1 consists of a struct containing the variable’s value and a forced-flag, which indicates the status of the variable. Before writing to the variable, the forced-flag is checked to ensure that it is cleared. This forced-flag is set every time the variable shall be overridden, and cleared whenever the overriding should stop. There is also a release all-function implemented in this concept, which overrides the forced-flags and releases all of the variables at once.

4.2.2 RTDB v2 - The Concept of a Real-Time Database consisting of Enumerated Lists and Local Lists of Variables

Our second concept, RTDB v2, is based on the idea of storing all variables in lists that are stored locally in one single module. This is similar to the way the real-time database is implemented in Coordinator 7, which is the ECU that handles the routing of CAN-buses in the vehicle. There is one list for each variable type, and each list has an enumerated list associated with it. The enumerated lists have the same amount of elements as the variable lists, where each element in the enumerated lists represents an index to the corresponding element in the variable lists.

The real-time database-module also consists of a read-, write-, force- and release-function for each variable type, similar to Scania’s implementation of RTDB in EEC3. These functions take the element in the enumerated lists as one of the input arguments and then use this element as an index to locate the corresponding variable in the variable lists. Since the variable lists are stored locally, the only way of accessing the variables in the database is through these functions. The direction of the data flow in this solution is illustrated in the example scenario in figure 4.6.

Similarly to RTDB v1, this concept is based on having a struct for each variable that contains the variable’s value and a forced-flag, which indicates the status of the variable. The forced-flag is set by the force-functions whenever the variable’s value should be overridden, and cleared by the release-functions whenever the overriding should stop. There is also a release all-function in RTDB, which overrides the forced-flags and releases all of the variables at once.
Figure 4.6: An illustration of RTDB v2, which is a real-time database consisting of enumerated lists and local lists of variables. There is a set of read-, write-, force- and release-functions for each variable type. In this example, there are two applications reading from three managers through the real-time database, where the variables are of two different variable types. The arrows represent the direction of the data flow, where dashed arrows represent hidden data flow.

4.2.3 RTDB v3 - The Concept of a Real-Time Database consisting of Specific Functions for Each Variable and Action

Instead of having a centralized architecture for the real-time database, our third concept, RTDB v3, is based on the idea of having the variables stored in one module for each software layer in the platform architecture. These modules contain a read-, write-, force- and release-function for each variable instead of each variable type. This concept is similar to the way Scania used to implement the real-time database in version S7 of the Engine Management System ECU. Some of the features with RTDB v3 are that it contains no global variable and no pointers. The direction of the data flow in this solution is illustrated in the example scenario in figure 4.7. The mechanism for the override functionality is similar to the mechanism in RTDB v2, where each variable is consisting of a struct containing a forced-flag. There is also a release all-function implemented in this concept, which releases all of the variables at once.
4.2.4 RTDB v4 - The Concept of a Real-Time Database consisting of one Specific Function for Each Variable

In our fourth concept, RTDB v4, the concept of RTDB v3 is developed further by merging all the read-, write-, force- and release-functions into one specific function for each variable. In this way the number of functions is decreased by 75% and all the hidden data flow is removed. The specific functions take two input arguments; one with the value that should be written to the variable, and one which specifies which variable action that should be performed. If the variable action is read or release, the input value is ignored. If the variable action is write or force, the return value of the function shall be ignored. The direction of the data flow in this solution is illustrated in the example scenario in figure 4.8. The mechanism for the override functionality is similar to the mechanism in RTDB v2, where each variable is consisting of a struct containing a forced-flag. There is also a release all-function implemented in this concept, which releases all of the variables at once.
Figure 4.8: An illustration of RTDB v4, which is a real-time database consisting of one specific function for each variable. The function handles the read-, write-, force- and release-action for each variable. In this example, there are two applications reading from three different managers through the real-time database, where the variables are of two different variable types. The arrows represent the direction of the data flow.

4.2.5 Testbed for our Concepts of the Real-Time Database

The functionality of the real-time database concepts presented in this section has been verified using a virtual testbed. This testbed was built using the gcc compiler and was executed on a Windows OS computer. It was built in order to simulate the data transfer between a sensor driver in the LLAP-layer and an application in the APPL-layer through the real-time database.

While the sensor driver was continuously writing a new value to the variable in the database, the application performed the variable actions read, force and release. To verify that the functionality of these functions was working, the application printed out the value of the variable to the terminal. Whenever the forced-action was performed, the write-action from the sensor driver did not affect the variable, until the released-action was performed. This ensures a good possibility for having our prototype concepts working in an actual ECU.
5. Measurement Tool

This chapter presents a measurement tool that has been developed for measuring the complexity of a software architectural design in terms of coupling and cohesion.

5.1 Measuring Complexity of a Software Architectural Design

A measurement tool was developed in the thesis project in order to measure the complexity of a software architecture in terms of coupling and cohesion. The tool measures the software quality metrics coupling and cohesion from an XML representation of the source code, which is provided by the thesis work from [19].

The XML-representation is created from a parser that extracts the structure of the source code, so that the measurement method described in subsection 2.3.6 can be applied.

In order to obtain a graphical representation of the source code that captures the coupling and cohesion attributes, we have formulated and implemented a measurement protocol. A measurement protocol is defined as the following [1]:

"A measurement protocol is a set of conditions that assures consistent repeatable measurements of an attribute."

The set of conditions that are used to define the measurement protocol are applied to the XML-representation from [19], and extracts the relevant information to a new XML-representation. The new representation abstracts the coupling- and cohesion attributes in a graph, so that the attributes can be quantified numerically according to the measurement instrument introduced by [1]. A measurement instrument is defined as the following [1]:

"A measurement instrument is a tool for mapping an abstraction of software to a number or category."

The outcome of the measurement instrument is numerical values of coupling and cohesion for a specific module in the software architecture. The work flow of the measurement tool is illustrated in the flow chart in figure 5.1.

---

1XML is a mark-up language for encoding documents and files in way that is both human- and machine-readable.

2A parser is a computer program that produces a structural representation of the input data using formal grammar.
The measurement tool consists of both the measurement protocol and the measurement instrument and can be separated into two parts. These parts are described in subsections 5.1.1 and 5.1.2 respectively. The tool is implemented in the programming language Python v2.7 with the supplementary libraries LXML and SciPy. LXML is a library that supports parsing of XML-files in order to use xpath queries, which is a language for addressing parts of an XML-document [5]. It also provides functions for creating new XML-files following a given name space. SciPy is a library that provides scientific functionality such as usage of matrices and matrix operations.

5.1.1 Measurement Protocol for the Metrics Coupling and Cohesion

The measurement protocol takes the XML-representation of the source code from [19] and extracts the relevant information for the coupling- and cohesion measurement to a new XML-representation. The structure of the XML-representation from [19] is illustrated in figure A.1 in Appendix A and the resulting XML-structure from the measurement protocol is illustrated in figure 5.2a. The Document Type Definition (DTD), which describes the structure of the XML-representation is presented in figure 5.2b.

The new XML-representation of the source code is used for producing a graph, which illustrates the source code as modules, nodes and edges. The modules in the graph correspond to software modules, and the nodes correspond to functions within software modules. The edges in the graph represent the dependency between
Figure 5.2: The resulting XML-structure from the measurement protocol is illustrated in (a) and the associated Document Type Definition (DTD) is presented in (b).

two functions within a module, or between two modules. The dependency between two functions (or modules) corresponds to data flow, which occurs when one of the following actions is performed:

a) Function $f_1$ assigns a value to a global variable, and function $f_2$ uses this value. Function $f_2$ is therefore dependent on $f_1$, hence $f_1 \rightarrow f_2$.

b) Function $f_1$ assigns a local variable the return value from function $f_2$. Function $f_1$ is therefore dependent on $f_2$, hence $f_1 \rightarrow f_2$.

c) Function $f_1$ calls function $f_2$ with a parameter that is used in $f_2$. Function $f_2$ is therefore dependent on $f_1$, hence $f_1 \leftarrow f_2$.

d) Function $f_1$ calls function $f_2$ with parameter that is a reference to a variable, and $f_2$ changes the value of the variable through the reference. Function $f_1$ is therefore dependent on $f_2$, hence $f_1 \leftarrow f_2$.

From the resulting graph $G$ where all the data dependencies are captured, a measurement graph $MG$ is extracted. This measurement graph illustrates the data dependencies for the module that is subject for the coupling- and cohesion measurement. The extraction is done by tracing all nodes and edges along the incoming paths for each node in the module. This is repeated for each node along the paths so that all dependencies for the module are included. The measurement graph $MG$ is the result of the measurement protocol and is used by the measurement instrument.
5.1.2 Measurement Instrument for the Metrics Coupling and Cohesion

From the XML-representation of the measurement graph $MG$, the measurement instrument maps $MG$ to two node × edge tables, representing the coupling and cohesion subgraphs $S$ and $S'$ respectively. From these tables, the coupling and cohesion is calculated according to the algorithm in subsection 2.3.6.

The algorithm for calculating coupling and cohesion is implemented in the programming language Python according to pseudo algorithms 5 and 6. The algorithm for calculating cohesion has all the functionality the coupling algorithm has, with the additional functionality of normalizing cohesion in the interval [0, 1].

Algorithm 5 Algorithm for the computation of coupling

1: procedure computeCoupling($S$)  
2:     $M = \text{createNodeEdge}(S)$  
3:     $I = \text{computeInformation}(S)$  
4:     for each row in $M$ do  
5:         $SM = \text{createSubTable}(row,M)$  
6:         $SI = SI + \text{computeInformation}(SM)$  
7:     end for  
8:     return $SI - I$  
9: end procedure

Algorithm 6 Algorithm for the computation of cohesion

1: procedure computeCohesion($S'$)  
2:     $M = \text{createNodeEdge}(S')$  
3:     maxM = \text{createMaxNodeEdge}(S')  
4:     $I = \text{computeInformation}(M)$  
5:     maxI = \text{computeInformation}(maxM)  
6:     for each row in $M$ do  
7:         $S'M = \text{createSubTable}(row,M)$  
8:         maxS'M = \text{createSubTable}(row,maxM)  
9:         $S'I = S'I + \text{computeInformation}(S'M)$  
10:        maxS'I = maxS'I + \text{computeInformation}(maxS'M)$  
11:    end for  
12:    return $(S'I - I)/(maxS'I - maxI)$  
13: end procedure

\text{createNodeEdge} converts the XML-representation into a matrix representing the node × edge table described in subsection 2.3.6.

\text{createMaxNodeEdge} creates a matrix representing a node × edge table where all nodes in the input XML-structure are connected to each other.
COMPUTE_INFORMATION computes the probability distribution of distinct row patterns in the node × edge table.

CREATE_SUB_TABLE creates a matrix that represents a subgraph consisting of only the edges that are directly connected to the node.

5.2 Testing Procedure for the Measurement Tool

The testing procedure for the measurement tool was divided into three steps; one for testing the measurement protocol, one for testing the measurement instrument and one for testing the complete measurement tool.

In order to test the measurement protocol, a small test program was developed. The test program contained all actions in subsection 5.1.1 for making data flow occurring in the source code. The output from the measurement protocol, the new XML-representation, was verified manually.

The measurement instrument was tested using a manually compiled XML-file, with the same format as the output from the measurement protocol. This XML-file was designed as one of the example graphs in [1]. This was done in order to compare the results of the calculations from the measurement instrument to the results in [1]. The example graph in [1] is illustrated in figure 5.3.

Finally, in order to test the complete measurement tool, the modular structure in figure 5.3 was realized in source code. The outcome of the tool was verified by a comparison with the calculated results in [1].

Figure 5.3: Example graph of a modular system from [1] used in the testing of the measurement instrument.
6. Verification of the SW Architecture

In this chapter we formulate the software safety requirements for the pressure sensor driver in EEC3 in a structured way using contract theory. This is used for verifying the software architectural design.

6.1 Verifying the Software Architectural Design for the Pressure Sensor Driver in EEC3

ISO 26262 states in requirement 7.4.2-a that the software architectural design shall be verifiable, so that a bi-directional traceability between the software architectural design and the software safety requirements can be achieved. Also, requirement 7.4.9 states that the software safety requirements shall be allocated to the software components. These two requirements are not fulfilled in Scania’s current implementation of the pressure sensor driver in EEC3 according to the analysis in section 3.1. A structural way of formulating the software safety requirements is therefore presented in section 6.1.1 so that the requirements from ISO 26262 can be fulfilled.

6.1.1 Formulating Software Safety Requirements

One of the benefits with formulating software safety requirements in a structured and formal way is the possibility of automating the process of verifying the software architecture. To achieve this, the requirements are formulated according to the method described in subsection 2.3.7, which is based on contract theory. We will demonstrate how the method is applied to the software architectural design of our third solution for the pressure sensor driver in EEC3, Pres v3. The reason for choosing this solution is because it has the least amount of dependencies to other software units in the system, which simplifies the formulation of the requirements. Since Pres v2 and Pres v3 have the same modular structure, the software safety requirements will be applicable to Pres v2 as well.

A block diagram for Pres v3 is illustrated in figure 6.1. The block diagram consists of the pressure sensor driver pres_v3 with all software units it is dependent on. In this case it is only the internal sensor driver intv, which provides the status of the battery supply voltage to pres_v3. When formulating the software safety requirements for pres_v3, all hardware is considered as the environment. Since we are considering all individual software units from an environmental-centric point of view, assumptions can be made on the functionality in the environment. A more extensive example of how to formulate functional safety requirements in a structured way is presented in Appendix B, where requirements allocated to the hardware components are formulated as well.
Formulating software safety requirements for the internal sensor driver

We will start with formulating contract $C_{\text{intv}}$ between the internal sensor driver $\text{intv}$ and the software units depending on $\text{intv}$. The internal sensor driver $\text{intv}$ provides the status of the battery supply voltage as the Boolean variable $B_{\text{bat}}$. This is to ensure that the battery voltage is at an operational level. The requirement $R_{1:1}$ for $\text{intv}$ is to provide the signal $B_{\text{bat}} = \text{TRUE}$ if $V_{\text{bat}} \in (16\,\text{V}, 32\,\text{V})$ and $B_{\text{bat}} = \text{FALSE}$ otherwise. This is formulated as the logical expression (6.1), where $V_{\text{bat}}$ is the actual battery voltage.

$$R_{1:1} : B_{\text{bat}} = (V_{\text{bat}} > 16\,\text{V}) \land (V_{\text{bat}} < 32\,\text{V}) \quad (6.1)$$

However, the software unit $\text{intv}$ does not have direct access to the actual battery voltage $V_{\text{bat}}$. It only has access to the sampled signal $S_{\text{bat}}$ from the A/D-converter. The requirement $R_{1:1}$ for $\text{intv}$ is therefore only fulfilled under the assumption that $S_{\text{bat}}$ corresponds to the actual battery voltage $V_{\text{bat}}$. This assumption is expressed in (6.2) and is allocated as a requirement to a component in the environment or to the environment itself, where $f_1$ is the transfer function for the hardware.

$$A_{1:1} : S_{\text{bat}} = f_1(V_{\text{bat}}) \quad (6.2)$$
As \( B_{bat} \) shall represent \( V_{bat} \) under the assumption that \( S_{bat} = f_{1}(V_{bat}) \), the implied behavior of the software unit \( \text{intv} \) is \( f_{1}^{-1}(\cdot) \). Hence:

\[
B_{bat} = f_{1}^{-1}(f(V_{bat})) = V_{bat}.
\]

The contract \( C_{\text{intv}} \), which consists of assumption \( A_{1:1} \) and requirement \( R_{1:1} \), is presented in table 6.1. This is according to the structure introduced by the contract theory in [31]. The ASIL classification of the requirements is stated in the contract to indicate which ASIL the requirement shall fulfill so that requirement 7.4.9 from ISO 26262 is fulfilled. The ASIL classification for \( R_{1:1} \) is C, since it is the level Scania is aiming to fulfill for this subsystem.

Table 6.1: Contract for the internal sensor driver \( \text{intv} \).

<table>
<thead>
<tr>
<th></th>
<th>( C_{\text{intv}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{1} )</td>
<td>( S_{bat} = f_{1}(V_{bat}) )</td>
</tr>
<tr>
<td>( R_{1} )</td>
<td>( B_{bat} = (f_{1}(V_{bat}) &gt; 16V) \land (f_{1}(V_{bat}) &lt; 32V) )</td>
</tr>
</tbody>
</table>

Formulating software safety requirements for the pressure sensor driver

The functionality of the pressure sensor driver \( \text{pres\_v3} \) is to reproduce the pressure value \( P_{\text{real}} \) at the pressure sensor, and provide it to device manager \( \text{devm} \) together with its associated service status. The service status, \( P_{\text{status}} \), can obtain any of the statuses: \( \text{NOM} \), \( \text{DIST} \) and \( \text{UNA} \). However, in this demonstration we will simplify \( P_{\text{status}} \) to only obtain the statuses: \( \text{OK} \) and \( \neg \text{OK} \). The status will be \( \neg \text{OK} \) whenever \( B_{bat} = \text{FALSE} \) or if the sampled pressure value \( S_{\text{pres}} \notin (0.175V, 4.6V) \). This is to ensure that the signal \( S_{\text{pres}} \) is not shorted to ground, the battery or an open load (i.e. disconnected). \( P_{\text{status}} \) can be formulated as the logical expression (6.3).

\[
(P_{\text{status}} = \text{OK}) = \neg B_{bat} \lor ((S_{\text{pres}} > 0.175V) \land (S_{\text{pres}} < 4.6V)) \quad (6.3)
\]

Figure 6.2 illustrates all combinations that can occur for the pressure value \( P_{\text{devm}} \) and the service status \( P_{\text{status}} \). The green states represent the states that are in compliance with ISO 26262 and the red state represents the state that is prohibited by ISO 26262. To avoid this fail state where \( P_{\text{devm}} = \neg P_{\text{real}} \) at the same time as \( P_{\text{status}} = \text{OK} \), the software safety requirement for \( \text{pres\_v3} \) is formulated as \( R_{2:1} \) in (6.4).

\[
R_{2:1} : (P_{\text{devm}} = P_{\text{real}}) \lor (P_{\text{status}} = \neg \text{OK}) \quad (6.4)
\]
Figure 6.2: All states that can occur for the pressure signal $P_{devm}$ and service status $P_{status}$, where $P_{real}$ corresponds to the actual pressure value at the pressure sensor. Only the green states are in compliance with ISO 26262.

However, since pres_v3 does not have direct access to the actual pressure value $P_{real}$, the requirement $R_{2:1}$ is only fulfilled under the assumptions $A_{2:1}$ and $A_{2:2}$ expressed in (6.5) and (6.6) respectively. Assumption $A_{2:1}$ is fulfilled by requirement $R_{1:1}$ from intv. In $A_{2:1}$ we assume that the sampled signal $S_{pres}$ reflects the actual pressure value $P_{real}$ at port $p_{12}$. This assumption is considered to be fulfilled by the environment, where $f_2$ is the transfer function for the hardware including the pressure sensor.

$$A_{2:1} : B_{bat} = (f_1(V_{bat}) > 16V) \land (f_1(V_{bat}) < 32V) \quad (6.5)$$

$$A_{2:2} : S_{pres} = f_2(P_{real}) \quad (6.6)$$

As $P_{devm}$ shall represent $P_{real}$ under the assumption that $S_{pres} = f_2(P_{real})$, the implied behavior for the software unit pres_v3 is $f_2^{-1}()$, in this ideal example. Hence:

$$P_{devm} = f_2^{-1}(f_2(P_{real})) = P_{real}.$$  

For this example to be complete, compensation for deviations due to e.g. quantification errors needs to be considered as well.

The contract $C_{pres\_v3}$ between pres_v3 and the software units depending on pres_v3, consists of assumptions $A_{1:1}$, $A_{1:2}$ and requirement $R_{2:1}$ and is presented in table 6.2. The ASIL classification for $R_{2:1}$ is C, since it is the level Scania is aiming to fulfill for this subsystem as well.

Table 6.2: Contract for the pressure sensor driver pres_v3.

<table>
<thead>
<tr>
<th>$C_{pres_v3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2$</td>
</tr>
<tr>
<td>1 $B_{bat} = (f_1(V_{bat}) &gt; 16V) \land (f_1(V_{bat}) &lt; 32V)$</td>
</tr>
<tr>
<td>2 $S_{pres} = f_2(P_{real})$</td>
</tr>
<tr>
<td>$R_2$</td>
</tr>
<tr>
<td>1 $(P_{devm} = P_{real}) \lor (P_{status} = \neg OK)$</td>
</tr>
</tbody>
</table>
6.2 Verifying Consistency of the Software Safety Requirements

The software safety requirements need to be consistent with regards to formula $A \cap B \subseteq R$ illustrated in figure 2.5 in subsection 2.3.7. Here $A$ stands for assumption, $B$ for behavior and $R$ for requirement. To verify the consistency of the software safety requirements for the pressure sensor driver in EEC3, a directed graph for contracts $C_{pres\_v3}$ and $C_{intv}$ is derived. The directed graph is illustrated in figure 6.3 where the relations between the requirements and assumptions are shown. This can be used for verifying the consistency of the safety requirements and the fulfillment of formula $A \cap B \subseteq R$.

![Directed graph for verifying the consistency of the software safety requirements for the software architectural design of Pres v3.](image)

Figure 6.3: Directed graph for verifying the consistency of the software safety requirements for the software architectural design of Pres v3.
7. Results

This chapter presents the measurement results of the sensor driver implementations and estimated characteristics of the real-time database concepts.

7.1 Measurement Results of the Pressure Sensor Driver Implementations

Scania’s current implementation of the pressure sensor driver in EEC3 has been evaluated together with our prototype solutions. The software quality metrics coupling and cohesion have been measured with the measurement tool described in section 5.1. The number of code lines has been counted using the freeware software program LOC Counter GUI v2011.8.27.1 from [14], where the code lines in the pressure sensor driver and the modules it is depending on are counted. Any empty code lines or comments are excluded in the calculations. The measurement results are presented in figures 7.1a, 7.1b and 7.1c and reflect the software architectural design of the pressure sensor driver.

To evaluate the software unit design of the different implementations of the pressure sensor driver, the numbers of global variables and pointers have been counted manually. All of the modules that are used by the pressure sensor driver are included in the calculations. The usage of hidden data flow has also been calculated where the number of data transfers through hidden data flow has been counted manually. The results are presented in figures 7.2a, 7.2b and 7.2c and reflect the software unit design.

The performance of the pressure sensor drivers has been estimated in terms of execution time and memory usage. The execution time has only been estimated by counting the number of executed code lines for providing a pressure value. These have been counted manually where each code line is treated equally and function calls are treated as one operation. Even though this is a crude way of measuring the execution time, it still gives a hint of what the actual execution time will be.

The memory allocation on the stack has been calculated by manually adding together the size of the non-static variables that are declared for the longest trail of function calls. Overhead for each function call is also included in the calculations, since the return address of the function is stored on the stack as well. The static memory usage has been calculated by manually adding together the size of the static variables that are declared in the sensor driver and all the modules the sensor driver is dependent on. The results are presented in figure 7.3a, 7.3b and 7.3c and reflect the performance of the pressure sensor driver. A summary of all of the results is illustrated in the spider chart in figure 7.4.
Figure 7.1: Measurement results for the different implementations of the pressure sensor driver regarding the software architectural design. The coupling of the sensor driver is presented in (a) and the cohesion is presented in (b). The number of code lines for the pressure sensor driver and all of the modules it is dependent on is presented in (c).

50
Figure 7.2: Measurement results for the different implementations of the pressure sensor driver regarding the software unit design. The number of global variables is presented in (a) and the number of pointers is presented in (b). The number of data transfers through hidden data flow is presented in (c).
Figure 7.3: Estimated performance of the different implementations of the pressure sensor driver. The number of executed code lines for providing a pressure value is presented in (a) and the size of the memory allocation on the stack is presented in (b). The size of the static memory usage is presented in (c).
Summary of the Pressure Sensor Driver Implementations

Figure 7.4: In this spider chart our developed implementations of the pressure sensor driver are compared with Scania’s current implementation in EEC3, which is at 100%. The cohesion in this chart is inverted so that the goal is to minimize the area and to be as close as 0% as possible.
7.2 Estimated Characteristics for the Real-Time Database Concepts

Scania’s current implementation of the real-time database has been evaluated together with our prototype concepts. Since the prototype concepts have not been fully realized in source code, only an estimation of the characteristics has been calculated. To estimate the coupling and cohesion of our concepts with our measurement tool, an example scenario has been simulated directly in the XML-format.

The example scenario consists of three applications and one manager communicating with each other through 15 variables in the real-time database. The variables are of three different variable types. The manager writes to ten variables in the database, where five of them are read by the first application and the other five are read by the second application. These two applications write to one variable each, which are read by the third application. The third application is in turn writing to three variables that are read by the manager. The coupling and cohesion for the different real-time database concepts are presented in figures 7.5a and 7.5b respectively. In RTDB v1 the cohesion is undefined since the global variables are distributed throughout the whole system and are not gathered in one specific module.

In order to estimate the number of code lines our different concepts of the real-time database will require, a few assumptions have been taken. The first assumption is that there are 189 variables in the database, which is equal to the number of variables that are currently residing in RTDB in EEC3. The number of pointers in Scania’s implementation of RTDB has been estimated to be three times as many as the number of variables. In order for the solutions to be comparable with each other, we have assumed that there are three read-actions and one write-action performed for each variable in the database. The calculation results are presented in figure 7.5c, and gives an estimation of how many code lines the implementations will require.

To evaluate the software unit design of the different concepts of the real-time database, the numbers of global variables and pointers have been calculated manually under the same assumptions as for calculating the number of code lines. The number of data transfers through hidden data flow has also been calculated manually. The results are presented in figures 7.6a, 7.6b and 7.6c and reflect the software unit design of the real-time database concepts.

Similarly to the pressure sensor driver, the performance in terms of execution time and the memory usage has been estimated. The number of executed code lines is presented in figure 7.7a and the memory allocation on the stack is presented in 7.7b. The static memory usage is presented in 7.7c. A summary of the estimated characteristics of the real-time database concepts is illustrated in the spider chart in figure 7.8.
Figure 7.5: Estimated characteristics for the different implementations of the real-time database regarding the software architectural design. The coupling of the real-time database is presented in (a) and the cohesion is presented in (b). The cohesion in \textit{RTDB v1} is undefined and there is no cohesion in \textit{RTDB v4}. The number of code lines for the real-time database is presented in (c).
Figure 7.6: Estimated characteristics for the different implementations of the real-time database regarding the software unit design. The number of global variables is presented in (a) and the number of pointers is presented in (b). There are no global variables in RTDB v3 and RTDB v4, and there are no pointers in RTDB v1, RTDB v3 and RTDB v4. The number of times data is fetched or manipulated through hidden data flow is presented in (c). There is no hidden data flow in RTDB v4.
Figure 7.7: Estimated performance of the different implementations of the real-time database. The number of executed code lines for each variable action is presented in (a) and the size of the memory allocation on the stack is presented in (b). There is no memory allocation on the stack in \textit{RTDB v1}. The size of the static memory usage is presented in (c).
Figure 7.8: In this spider chart our developed concepts for the real-time database are compared with Scania’s current implementation in EEC3, which is at 100%. The cohesion in this chart is inverted so that the goal is to minimize the area and to be as close as 0% as possible. Since the cohesion in RTDB v1 is undefined and since there is no cohesion in RTDB v4, the cohesion of those two versions cannot be displayed in the chart.
8. Discussion and Recommendations

In this chapter the results of our different sensor driver and real-time database-concepts are discussed, as well as the software metrics used for evaluating these concepts. The compliance with ISO 26262 including the verification of the software safety requirements are also discussed. The chapter finishes with suggestions for improvements and future work.

8.1 Discussion about the Sensor Drivers in EEC3

The main idea of all our alternative concepts for the sensor drivers in EEC3 is to decrease the complexity of the software architecture while fulfilling the software safety requirements from ISO 26262. This has been done by modifying the internal structure of the sensor driver and avoiding the reusable software functions that reside in COMP and the UTLS-layer. The software architectural design and the software unit design of both Scania’s solution and our alternative solutions are discussed in subsections 8.1.1 and 8.1.2. The performance of our solutions is discussed in subsection 8.1.3. The formulation of the software safety requirements and the solutions compliance with ISO 26262 are discussed in subsections 8.1.4 and 8.1.5 respectively.

8.1.1 The Software Architectural Design of the Sensor Drivers in EEC3

In Scania’s architectural design of the sensor drivers in EEC3, the sensor drivers are coupled very tightly to other modules in the system. This is because a lot of the functionality is done through reusable software functions in COMP and the UTLS-layer. One of the benefits with reusable software is faster development time for new applications. Also, if the reused software is thoroughly tested it will reduce the risks of faults. However, since coupling reflects the number of code files the software developers need to examine and understand every time a change is made, reusable software functions might increase the time consumed by the software developers anyway.

The measurement results in figure 7.1a for our alternative pressure sensor driver solutions indicate that the more the reusable software functions are avoided, the looser the coupling becomes. Pres v1 has tighter coupling than Pres v2 and Pres v3 because of the dependencies to functions in the UTLS-layer. The reason why Scania’s current solution has the tightest coupling of all is because of the additional dependencies to functions in COMP. The only coupling occurring in Pres v2 and Pres v3 is because of the dependency to module intv, which is another sensor
driver that provides the status of the internal battery supply voltage. This kind of coupling is considered by the authors as justified.

Scania’s software architectural design of the pressure sensor driver has lower cohesion than all our alternative solutions, especially Pres v2 and Pres v3. Even though the internal structure of Scania’s solution and Pres v2 is similar, there is a significant difference in cohesion. This is because cohesion scales with the total number of functions the sensor driver is dependent on, according to the algorithm in [1]. This explains why Pres v1 and Pres v3, which have the same internal structure, differ a lot in cohesion. Pres v2 has the highest cohesion since the functionality of the sensor driver is separated into functions that perform a more limited task than the functions in Pres v1 and Pres v3 do.

Dividing the pressure sensor driver into two separate modules; one for the AdBlue fluid and one for the diesel fuel, decreases the complexity of the architectural design even further. Having them together causes logical cohesion, which makes the implementation more difficult to reuse and can cause maintenance problems, since the source code is intertwined. Separating the source code for the pressure sensor driver increases the number of code lines if the vehicle contains both a SCR-system and a DPF-system. However, since the sensor drivers work in the same way the software developer only needs to understand one of these sensor drivers in order to understand how the other one works.

The results in figure 7.1c illustrate how many code lines that needs to be read in order to understand the implementation of the pressure sensor driver. Even though figure 7.1c shows that Scania’s current solution requires more code lines than any of our alternative solutions, the total number of code lines will be lower in Scania’s solution if our concepts are applied to all sensor drivers in EEC3. This is because a lot of the software is reused in Scania’s current solution.

Having a decentralized service based architecture for fault-tolerant control reduces the complexity considerably, since the current centralized architecture of DIMA-BSW has grown to be so complex that it is difficult to understand and follow the source code. Distributing the fault-tolerant mechanism throughout the system is more likely to be easier to maintain, which can lead to reduced maintenance costs for Scania.

8.1.2 The Software Unit Design of the Sensor Drivers in EEC3

Scania’s current solution for the sensor drivers is based on having one fundamental struct associated with each sensor signal, where the struct contains pointers to global variables at file scope. Initiating the pointer values in the main-function of the sensor driver is something we consider as an example of poor software design. All initialization that only needs to be done once should be done during the initialization-phase and not during run-time, otherwise it will increase the execution time significantly. In the case of the pressure sensor driver, this would save thousands of executed code lines per second.
The concept of having a fundamental struct containing pointers for each sensor signal is however abandoned in all of our three alternative solutions. The only pointer that is kept is the one that connects the sensor driver with its associated channel in the A/D-converter. Limiting the number of pointers to only one per sensor driver reduces the complexity of the source code [27].

Most of the global variables in Scania’s current implementation of the sensor drivers are declared in the sensor drivers’ calibration file. Storing these variables in a separate calibration file is something we consider as justified usage of global variables, since it is appropriate to store the variables that are more prone to be changed by the software developer in a separate place. This is only under the condition that the variables are declared as constants so that they cannot be modified by the software itself.

The hidden data flow was reduced by abandoning the concept of a fundamental struct containing pointers to global variables at file scope. The only hidden data flow occurring in our alternative solutions for the pressure sensor driver is between the initialization-function and the main-function of the sensor driver. This is something that has not been addressed in any of our solutions, since initialization of the sensor drivers is out of scope for the thesis.

8.1.3 The Performance of the Sensor Drivers in EEC3

The results in figure 7.3a indicate that reusable software functions increases the execution time significantly. By comparing \textit{Pres v1} with \textit{Pres v3}, we can observe the impact the utilities-functions and the module structure have on the execution time. Moving the functionality of the utility-functions into the sensor driver module, and separating the pressure sensor driver into two sensor driver modules lead to approximately 50\% faster execution time.

By comparing \textit{Pres v1} with Scania’s current implementation of the pressure sensor driver, we can observe the impact the reusable software functions in COMP and the internal structure of the sensor drivers have on the execution time. Moving the functionality of the functions in COMP into the sensor driver and abandon the concept of a fundamental struct for each sensor signal, also lead to approximately 50\% faster execution time. By comparing \textit{Pres v2} with \textit{Pres v3} we can observe that the differences in the internal structure do not impact the execution time significantly.

The memory allocation on the stack is significantly higher in Scania’s current implementation of the pressure sensor driver than any of our alternative solutions. This is mainly because reusable software functions allocate more memory on the stack than specialized source code. \textit{Pres v1} allocates more memory on the stack than \textit{Pres v2} and \textit{Pres v3} because of the use of utility-functions. There is no difference between the memory allocation on the stack between \textit{Pres v2} and \textit{Pres v3}.

The static memory is divided into two types of memories; instruction memory and data memory. The size of the memory stored in the data memory is reduced with approximately 30\% in our alternative solutions compared with Scania’s current
implementation. This is mainly because the concept of having a fundamental struct containing pointers to global variables at file scope is abandoned.

The instruction memory is storing all the instructions that the micro-controller’s CPU is performing, and the memory allocated in the instruction memory is related to the number of code lines in the source code. Since our alternative solutions are avoiding reusable software functions, our alternative solutions will require more instruction memory if they are scaled to all of the sensor drivers in the ECU. However, according to Scania employee Andreas Rasmusson, both the instruction memory and the data memory are not considered to be an issue at Scania since the available space of these memories is very high.

8.1.4 The Formulation of Functional Software Safety Requirements for the Sensor Drivers in EEC3

For block diagrams consisting of several software units, it is important to check the consistency of the contracts in order to verify that the subset of requirements fulfills the functional requirement. A contract based structure has proven to simplify the verification process considerably as well as enabling the possibility of automating the verification process. This might reduce the software maintenance costs and increase the safety, since it facilitates identification of safety critical nodes in the software architecture.

During the process of formulating the functional software safety requirements, we found that low coupling and avoiding reusable software functions simplifies the formulation significantly. Since coupling reflects the complexity of the software architectural design, the formulation of the functional software safety requirements will also become more complex. The software architecture of Pres v2 and Pres v3 is therefore much easier to verify than Pres v1 and Scania’s current solution.

8.1.5 The Sensor Drivers’ Compliance with ISO 26262

In table 2.3 there are principles listed that the software architectural design shall exhibit. Many of these principles are not very specific, which makes it difficult to know whether the requirements are fulfilled or not. The only requirements that we consider could be improved in Scania’s software architectural design of the sensor drivers in EEC3, is having restricted coupling between software components and high cohesion within each software component. These two issues have been addressed in all of our alternative solutions, where Pres v2 has the best performance of those two metrics combined. All the other requirements that Scania’s software architecture is fulfilling regarding the principles in table 2.3 is also fulfilled in all of our alternative solutions.

Regarding the software unit design, all of our three alternative solutions fulfill the requirements better than Scania’s current implementation of the pressure sensor driver. The only requirements that could be improved in Scania’s current implementation are limiting the use of pointers and removing any hidden data flow. In our
concepts the number of pointers has been limited to only one per sensor driver. Re-
moving all the hidden data flow is only partially fulfilled in our alternative solutions
because of the hidden data flow that occurs during the initialization-phase.

In order to verify the software architectural design according to requirement
7.4.2-a in ISO 26262, it is considered to be of great importance to have a clear
definition of ports and relations between ports. This is where the framework for
contract theory can be applied. A formally defined contract for each software unit
simplifies the verification process and enables a bi-directional traceability between
the software architectural design and the software safety requirements. The formu-
lation of the software safety requirements for both Pres v2 and Pres v3 is therefore
fulfilling requirement 7.4.2-a from ISO 26262.

According to requirement 7.4.9 in ISO 26262, each software safety requirement
shall be allocated to the software components so that each software component is
developed in compliance with the highest ASIL allocated to it. Software safety re-
quirements have been allocated to the software components in Pres v2 and Pres v3,
which are developed in compliance with ASIL C. The requirement is therefore ful-
filled for both Pres v2 and Pres v3.

8.2 Discussion about the Real-Time Database in EEC3

Different concepts of the real-time database have been designed in order to evaluate
the concepts in terms of complexity, performance and compliance with ISO 26262.
The software architectural design and software unit design of Scania’s current
real-time database in EEC3 and our alternative solutions are discussed in sub-
sections 8.2.1 and 8.2.2. The performance of the different concepts is discussed
in subsection 8.2.3 and the compliance with the requirements from ISO 26262 is
discussed in subsection 8.2.4.

8.2.1 The Software Architectural Design of the Real-Time Database in EEC3

Scania’s current RTDB in EEC3 has relatively loose coupling in comparison to our
alternative concepts. The only concept that has looser coupling according to our
measurement tool is RTDB v1, which is based on a real-time database consisting
of global variables. This type of coupling is however classified as common coupling
and is considered as one of the tightest form of coupling according to table 2.3 in
subsection 2.3.4. Since the measurement algorithm in [11] that our measurement tool
is based on does not consider different types of coupling, the results in 7.5a may be
misleading.

The reason why Scania’s current RTDB in EEC3 and RTDB v2 are estimated
to have the same amount of coupling is because they both are based on the same
structure of functions. Having a function for each variable type and variable action,
limits the amount of functions and the number of intermodule edges. Since coupling
is proportional to the number of intermodule edges, the coupling in Scania’s current
RTDB in EEC3 is estimated to be more restricted than RTDB v3 and RTDB v4, where the intermodule edges are scaled with the number of variables in the database.

The cohesion in the real-time database in EEC3 is very low. This is because the module which handles the real-time database consists of functions that handle all the different variable types, and functions that handle different variable types do not interact with each other. To increase the cohesion, the module could be divided into one module for each variable type. It is however more practical to gather all these functions into one single module and this does not decrease the readability of the source code significantly.

The cohesion in RTDB v1 is undefined since the global variables are distributed throughout the whole system and are not gathered in one specific module. RTDB v2 is estimated to have the same amount of cohesion as Scania’s current RTDB in EEC3 since the internal structure is similar, and RTDB v3 is estimated to have even lower cohesion than that. The cohesion will decrease even further in RTDB v3 when it is realized in an ECU, since the cohesion scales with the number of variables in the database. RTDB v4 does not have any cohesion at all since there is no interaction between the functions within the modules.

The number of code lines is relatively low in Scania’s current RTDB in EEC3 in comparison to RTDB v3 and RTDB v4. This is because RTDB v3 and RTDB v4 scales faster with the number of variables in the database. However, the number of code lines does not affect the time consumed for software developers to understand the source code as much, since the software developers do not need to read and understand all of the code lines in order to understand how the database works.

8.2.2 The Software Unit Design of the Real-Time Database in EEC3

The number of global variables in the Scania’s current RTDB in EEC3 is significantly higher than any of our alternative concepts. This is mainly because of the pointers that points to the variables are declared as global. Since there are in average three pointers pointing at each variable, due to requests from different software developers, the number of global variables is approximately increased with 200%.

Having different names when referring to the same variable can however be a disadvantage since it increases the complexity and makes the source code more difficult to understand and follow. The main reason for having different names is to abstract details in the source code to developers who are not considered to need to understand those details. But if software developers understand more of the details in the source code, they are more likely to make better design decisions.

Limiting the use of pointers has been done in all of our alternative concepts for the real-time database. It is only RTDB v2 that has pointers in the form of enumerated lists, since they act as indexes to local lists of variables in the database. Hidden data flow is something that occurs in all of the different concepts of the real-time database, except for RTDB v4, which is specifically designed to address this issue.
8.2.3 The Performance of the Real-Time Database in EEC3

When analyzing the performance in terms of execution time and memory allocation on the stack, it is the variable actions `read` and `write` which are of most interest. This is because the variable actions `force` and `release` only are performed for testing purposes and in the aftermarket. The results in figure 7.7a show that `RTDB v1` is estimated to have the fastest execution time for all of the variable actions. The reason why `RTDB v2` has one more executed code line for each variable action, compared to the real-time database in EEC3, is because of a safety check that ensures that the index to the variable list is not out of range. `RTDB v4` is estimated to have the slowest execution time since the functions always have to start by identifying which variable action that is going to be performed.

The only concept that does not allocate any memory on the stack is `RTDB v1`, since the variables can be read and modified without having to go through a function. The reason why the read-action allocates more memory on the stack in Scania’s RTDB in EEC3 and `RTDB v2` is because it needs to declare a temporary variable every time a struct is read. This is because the concepts are fetching the variables using pointers, which only points at the first member of the struct.

The number of code lines reflects the instruction memory usage in the CPU and is estimated to be significantly higher in `RTDB v3` and `RTDB v4` than any of the other three concepts. The concept which requires the least amount of instruction memory is `RTDB v1`. In terms of data memory, Scania’s current solution in EEC3 requires more memory than all of the other concepts. This is mainly because of all the pointers that are declared for each variable in the database.

8.2.4 The Real-Time Database’s Compliance with ISO 26262

The main ideas for better compliance with ISO 26262 regarding the software architectural design are to restrict the coupling of the real-time database and increase the cohesion. According to our estimates, Scania’s RTDB in EEC3 and `RTDB v2` fulfills these requirements best. Even though the coupling is looser in `RTDB v1`, it has the type of coupling that is considered to be one of the tightest. Also, having global variables does not promote encapsulation, which is one of the properties the software architectural design shall exhibit. No conclusions can be drawn from `RTDB v1` regarding cohesion, since the cohesion of that concept is undefined.

The main ideas for better compliance with the requirements regarding the software unit design are to avoid global variables, limit the use of pointers, and remove the hidden data flow. `RTDB v4` is the only concept that fully complies with ISO 26262 since it has no global variables, no pointers and no hidden data flow. The concept that complies with ISO 26262 the worst regarding the software unit design is Scania’s RTDB in EEC3.

Requirements 7.4.2-a and 7.4.9 regarding the verification of the software architecture are still a subject for improvement, for the real-time database. This is because our concepts have not been fully realized in source code during this thesis.
8.3 Suggestions for Improvements and Future Work

This thesis is focusing on the software architectural design of the sensor drivers and real-time database. The complexity of the software architectural design of Pres v2 is lower than Scania’s current solution, while the performance in terms of execution time and memory usage is better. Since it also complies with the requirements from ISO 26262, this solution is suggested to be realized in EEC3.

The complexity of the software architectural design of RTDB v2 is equal to Scania’s current solution, while the software unit design complies better with the requirements from ISO 26262 than what Scania’s real-time database in EEC3 does. Even though RTDB v4 is the only one that fully complies with the requirements from ISO 26262 regarding the software unit design, it is at the expense of the software architectural design. It is therefore suggested to realize RTDB v2 in EEC3.

It is suggested to distribute the functionality of COMP throughout the system instead. This will reduce the complexity of the architecture, which in turn might lead to reduced software maintenance costs. The reusable software functions in the UTLS-layer could be made as templates instead. In this way the benefits with reusable software functions, such as faster development time for applications and thoroughly tested source code, are kept without compromising with the coupling, cohesion and performance.

To obtain more accurate measurement results from the measurement tool, the measurement protocol could be divided into different measurement protocols, one for each type of coupling and cohesion that can occur. During the quantification of these software quality metrics, the measurement instrument could weight these types according to the impact they have on software quality. To adapt the measurement tool better to Scania’s needs, the coupling and cohesion values could be normalized to the coupling and cohesion of Scania’s current software architectural design in EEC3. This would make it easier for the software developers to know whether or not their changes increase the quality of the software.

For the verification of the software safety requirements, it is suggested to formulate the requirements of the software units in a formal notation regarding the relationships between the software units. This will enable the possibility of automating the verification process. A formal notation would also enable the possibility for checking the consistency of the software safety requirements allocated to one or more software units. Also, it would have the possibility of identifying safety critical nodes in the software architectural design.

Even if the verification process is automated, the formulation of safety requirements is considered to be so complex that it would be difficult for a computer to formulate these based on the source code. This is because the software requirements reflect the software developers’ intentions, which is difficult for a computer to understand. If the requirements and assumptions are following a formal notation, it would be possible to automate two things. Firstly, the chain of assumptions and requirements could be verified so that it fulfills the software requirements. Secondly, it could be possible to generate test cases for the behavior of the implementation.
9. Conclusion

The results from this thesis are showing that reusing common software functions is increasing the complexity of the software architecture significantly. An alternative architectural design for the platform software in one of Scania’s safety related embedded systems has therefore been proposed in this Master’s thesis. The proposal is exemplified in two solutions denoted *Pres v2* and *RTDB v2*, which manages the pressure sensor driver and the real-time database in one of Scania’s ECUs. The solutions are based on specialized source code that avoids reusable software functions.

In addition to better compliance with the safety requirements from ISO 26262, our proposal decreases the time consumed for software developers to understand and debug the source code, which diminishes the software maintenance costs. Our proposed software architecture has looser coupling between software components and higher cohesion within each software component, without compromising with the essential functionality and performance. This simplifies maintenance, verifiability and the possibility of identifying safety critical nodes in the software architectural design.

To increase the accuracy of the measurement tool developed in the thesis project, it can be further developed to distinguish between the different types of coupling and cohesion that can occur. To increase the benefits of our proposed software architectural design, all parts of the platform software architecture in Scania’s safety related embedded systems can be further developed in accordance with our proposal.
Bibliography


A. XML-structure of the Source Code

Here is graphical view of the XML-structure used to represent source code. This has been developed in Oskar Molin’s thesis work [19].

Figure A.1: The structure of the XML-representation of the source code [19].
B. Functional Safety Requirements for a Temperature Sensor Driver in EEC3

This is a formulation of the functional safety requirements allocated to a temperature sensor driver in EEC3, where requirements allocated to the hardware components are formulated as well. The formulation was done in collaboration with Jesper Ulke [28] who was working in parallel with us.

\[ p_1 = T_{devn} \]
\[ p_2 = S_{temp} \]
\[ p_3 = S_{temp} \]
\[ p_4 = GND \]
\[ p_5 = V_{temp1} \]
\[ p_6 = V_{temp1} \]
\[ p_7 = GND \]
\[ p_8 = V_{temp2}, I_{temp2} \]
\[ p_9 = V_{supply} \]
\[ p_{10} = V_{temp2}, I_{temp2} \]
\[ p_{11} = GND \]
\[ p_{12} = T_{real} \]
\[ p_{13} = GND \]
\[ p_{14} = T_{real} \]
\[ p_{15} = V_{supply} \]

Figure B.1: Block diagram for the temperature sensor driver tmpa.
Table B.1: Contract for tmpa.

<table>
<thead>
<tr>
<th>C_{tmpa}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>R_1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Table B.2: Contract for the A/D-converter.

<table>
<thead>
<tr>
<th>C_{A/D-converter}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>R_2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Table B.3: Contract for the analog hardware.

<table>
<thead>
<tr>
<th>C_{analog_hw}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>R_3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Table B.4: Contract for the temperature sensor.

<table>
<thead>
<tr>
<th>C_{temperature_sensor}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_4</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>R_4</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Figure B.2: Directed graph for verifying the consistency of the functional safety requirements for the software architectural design of tmpa.