TOWARDS AUTOMATED RECOVERY OF EMBEDDED SYSTEM FUNCTIONAL ARCHITECTURE
FROM SOURCE CODE AND PRODUCT DATA

OUSSAMA CHAMMAM
AHMED ZAMOUCHE

Master of Science Thesis
Stockholm, Sweden 2013
Towards automated recovery of embedded system functional architecture
from source code and product data

Oussama Chamamam
Ahmed Zamouche

Master of Science Thesis MMK 2013:45 MES 004
KTH Industrial Engineering and Management
Machine Design
SE-100 44 STOCKHOLM
För ett automatiserat återskapande av inbyggda systems funktionella arkitektur från källkod och produkt data

Ahmed Zamouche
Oussama Chammam

Sammanfattning

"För ett automatiserat återskapande av inbyggda systems funktionella arkitektur från källkod och produkt data"

Den ökade komplexiteten i inbyggda system inom fordonsindustrin tillsammans med de striktare säkerhetsrestriktionerna som infördes av ISO26262 standarden, kräver bättre kunskap och kändedom om produktarkitekturen. Men, för befintliga produkter som inte var utvecklade enligt en väldefinierad arkitekturmodell, så måste en modell återhämtas.

Syftet med detta examensarbete är att automatisera återhämtningen av funktionella arkitekturen för fordons inbyggda system, vilket är ett krav för många av ISO26262 aktiviteter. Detta examensarbete föreslår och beskriver två modeller för det inbyggda systemet i ett fordon, och visar dess användning för att bland annat generera användarvänliga vyer. Återhämtningen av modellerna sker genom att tolka den inbyggda C-koden och bearbeta fordonets data såsom inblandade styrenheter, deras adresser och CAN buss detaljer.


De föreslagna modellerna har framgångsrikt använts för att generera funktionella arkitekturen för ett par SCANIA lastbilar. Generering och återhämtningen av modellerna utfördes med hjälp av ett verktyg som utvecklades för detta ändamål. Vidare så har en standardiseringsmekanism från de föreslagna modellerna till AUTOSAR också tagits fram och presenterats. Standardiseringsmekanismen är rättfram när styrenhets krängutrustning inte beaktas i modellen.

I framtiden bör sensorer och ställdon inkluderas i modellerna. En mer detaljerad studie av den inbyggda programvaruarkitektur modellen, beträffande databeroende, bör också genomföras för att ta itu med problemen rörande felaktiga data-flödesvägar vilka har träffats på i detta arbete. Dessa problem uppkommer vid steget för CAN buss abstraktion.
“Towards automated recovery of embedded system functional architecture from source code and product data”

The increased embedded system complexity in the automotive industry together with stricter safety constraints introduced by the ISO26262 standard, require a better knowledge about the product architecture. However, for existing products which were not developed according to a well-defined architecture model, the latter need to be recovered.

The objective of this thesis work is to automate the recovery of the functional architecture in a vehicle, which is required for many of ISO26262 activities. The work of this thesis proposes and describes two embedded system models for the target system, and shows their usage to generate user friendly views. The recovery of the models is done by parsing embedded C-code and fetching vehicle's data such as involved ECUs, their addresses and CAN bus details.

This work has proposed two models for capturing the recreated information about an automotive embedded system: a product model for the embedded system and an architecture model for the embedded software. The product model is a simple embedded system model that only includes needed hardware and software details for the task of generating the functional architecture. The embedded software architecture model is derived from the product model and abstracts all hardware information. The embedded software architecture model covers only the high-level component-based software in all ECUs together abstracting away allocation and CAN bus information.

The proposed models have been successfully used to generate functional architecture for a couple of SCANIA trucks. The generation and recovery of the models was performed by a software tool that has been developed for this purpose. In addition, a mapping from the embedded software model to AUTOSAR standard has been proposed as a way to standardise the representation. The mapping to AUTOSAR showed that it is quite straight forward when not taking in consideration any possible ECU peripherals.

In the future, representation of sensors and actuators should be included in the models. A more detailed study of the architecture model for the embedded software, with regards to data-flow, should also be conducted to tackle issues related to wrong data-flow paths which have been found in this thesis. The issues arise in the step of CAN bus abstraction.
تحوي أطروحة الماجستير في العلوم 400 من مصدر وبيانات المنتج

العنوان: نموذج استخلاص البنية الوظيفية للنظام المضمون

النظام المضمن

أحمد زعموش

أسامة شمام

الموجز

تحوي أطروحة استخلاص البنية الوظيفية للنظام المضمون من مصدر وبيانات المنتج، إن الزيادة في تعقيد النظام المضمون في السيارات بالإضافة إلى شروط السلامة الأكثر صرامة والتي أدخلها معيار ISO 26262، تتطلب معرفة أفضل بينية المنتج. ولذلك، فبالنسبة للسيارات التي تم إنتاجها بدون تتبع نموذج معماري واضح العالم، فإن هذا الأخير يجب أن يتم استخلاصه لا حقا.

الهدف من هذه الأطروحة إثبات كفاءة أطروحة استخلاص البنية الوظيفية للنظام المضمون في السيارة، وهو الأمر المطلوب للعديد من أنظمة المعيار ISO 26262. هذه الأطروحة تفترض نموذج للنظام المضمون، وتظهر كفاءة استخدامها لخلق عروض حاسوبية سهلة الفهم والاستعمال. يتم استخلاص النموذج عبر تحليل مصدر لغة البرمجية C وبيانات السيارة المختلفة من أنظمة التحكم المتاحة، عناوينها وتفاصيل الشبكة (CAN).

يقترح هذا العمل نموذجين جزئيين للنظام المضمون في السيارات: نموذج المنتج للنظام المضمون ونموذج البرمجيات للنظام المضمون فقط. النموذج المنتج هو نموذج مبسط للنظام المضمون يحتوي فقط على الحد الأدنى من التفاصيل اللازمة عن الأجهزة والبرمجيات التي يقوم بها النظام. النموذج البرمجيات للنظام المضمون يحتوي على النموذج المنتج، وهو يعكس فقط البرمجيات المتاحة في الشبكات المتاحة في أنظمة التحكم كنها في عدم واحد. بذا القول، فإن هذا النموذج لا يجب أن يحتوي أي معلومات عن الأجهزة بما في ذلك معلومات موقع البرامج في أجهزة التحكم وتعاليم الشبكة (CAN).

لم يتم استخدام النماذج المقترحة بنجاج لتحويل بنية وظيفية لبعض شائعات شائعة التي تتطلب تحليل مصدر لغة البرمجية C، ولكنهم أظهرت أن النموذج النموذج المصغر فيه عملية بسيطة ومعتمدة إذا لم يأخذ في الاعتبار أي أجهزة مرجعية قد تكون مرتبطة بأجهزة التحكم. مستقبلاً، ينبغي إضفاء التعديلات المميزة والمشتمل على النماذج المقترحة، كما ينبغي إجراء دراسة أكثر تفصيلاً لنموذج البرمجيات للنظام المضمون فيما يتعلق بتلك البيانات، وذلك لمعالجة بعض الإشكالات المتعلقة بمصادر تدفق خاطئة للبيانات، والتي تم العثور عليها في هذه الأطروحة. هذه الأخطاء تنشأ في خطوة تجريد معلومات شبكة (CAN).
This Chapter contains a foreword written by the authors of this thesis work.

We would like to thank our supervisors, Professor Mattias Nyberg at SCANIA and associate Professor De-Jiu Chen from KTH, for their good advice and support.

We would also like to extend a thank you to SCANIA CV AB in Södertälje that opened their doors to us and gave us the opportunity to do this thesis.

I, Oussama would like to thank Allah (God) that made this moment to a reality and lead me gently towards the goal of conducting and finishing my thesis. I would also like to thank my parents and my family for their appreciated help through the whole five years of my university studies, which made it possible for me to reach this turning point in my life. Thank you Ahmed for the good time we had together during the last two years, and especially for the last five months where I have got a chance to know you more. I have learned much from you; a pleasant and a hard working friend!

I, Ahmed would like to thank my wife for her support during my master studies at KTH. I would also like to thank my colleagues at KTH for the fruitful discussions during our studies. I would like to thank my colleague and partner Oussama for his patience and creativity during the past months.

Oussama Chammam and Ahmed Zamouche

Stockholm, June 2013
NOMENCLATURE

Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Edge</td>
</tr>
<tr>
<td>v</td>
<td>Vertex</td>
</tr>
</tbody>
</table>

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Application Component</td>
</tr>
<tr>
<td>AE</td>
<td>Allocation Element</td>
</tr>
<tr>
<td>ARXML</td>
<td>AUTOSAR schema</td>
</tr>
<tr>
<td>AROS</td>
<td>High-performance transaction server</td>
</tr>
<tr>
<td>ATS</td>
<td>Aros Transaction Server, makes it possible to use SQL with Rosam databases.</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>AUTomotive Open System ARchitecture</td>
</tr>
<tr>
<td>AWS</td>
<td>Aros Work Station, a mainframe terminal emulator.</td>
</tr>
<tr>
<td>BCI</td>
<td>Bodywork Communication Interface</td>
</tr>
<tr>
<td>BSW</td>
<td>Basic Software</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>COO</td>
<td>Coordinator System</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DEC</td>
<td>Discrete Electric Circuit</td>
</tr>
<tr>
<td>DFG</td>
<td>Data Flow Graph</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>ESPRESSO</td>
<td>A project within SCANIA R&amp;D department</td>
</tr>
<tr>
<td>FAD</td>
<td>Functional Allocation Description</td>
</tr>
<tr>
<td>FV</td>
<td>Function Variable</td>
</tr>
<tr>
<td>GBM</td>
<td>Gearbox Management System</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>ID</td>
<td>IDentifiers</td>
</tr>
<tr>
<td>Midd</td>
<td>Midd Layer</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit</td>
</tr>
</tbody>
</table>
**PF**  PDU Format

**PGN**  Parameter Group Number

**PPORT**  Providing Port

**PS**  PDU Specific

**ROSAM**  A database management system

**RPORT**  Requires Port

**RTDB**  Real Time Database

**RTE**  Real Time Environment

**SESAMM**  SCANIA Electrical System

**SW-C**  Software Components

**UF**  User Function

**VFB**  Virtual Functional Bus

**VIN**  Vehicle Identification Number

**XML**  Extensible Mark-up Language

---

**Keywords**

# Table of Contents

**Sammanfattning** ..................................................................................................................... 2
**Abstract** .................................................................................................................................... 4
**الموجز** .................................................................................................................................... 6
**FOREWORD** ............................................................................................................................... 8
**NOMENCLATURE** ....................................................................................................................... 10

## 1 INTRODUCTION

1.1 Background and motivation .................................................................................................. 14
1.2 Purpose .................................................................................................................................... 14
1.3 Solution concept ..................................................................................................................... 15
1.4 Approach ................................................................................................................................ 15
1.5 Delimitations .......................................................................................................................... 17
1.6 Definitions .............................................................................................................................. 17
1.7 Division of work ..................................................................................................................... 18

## 2 RELATED TECHNOLOGIES, METHODS AND STANDARDS

2.1 Abstraction levels as defined in EAST-ADL2 ........................................................................ 20
2.2 Target system architecture ................................................................................................... 21
   2.2.1 Physical structure ........................................................................................................... 21
   2.2.2 CAN communication ..................................................................................................... 21
   2.2.3 Terminology and definitions .......................................................................................... 23
   2.2.4 Software structure ......................................................................................................... 24
      2.2.4.1 Application components ......................................................................................... 25
2.3 ISO26262.................................................................................................................................. 25
2.4 AUTOSAR............................................................................................................................... 27
   2.4.1 Software architecture ..................................................................................................... 27
      2.4.1.1 Basic software layer ............................................................................................... 27
      2.4.1.2 RTE ....................................................................................................................... 27
      2.4.1.3 Application layer .................................................................................................... 28
   2.4.2 Application interfaces ..................................................................................................... 30
   2.4.3 AUTOSAR methodology ............................................................................................... 30
2.5 Graph theory .......................................................................................................................... 31
   2.5.1 Hypergraphs .................................................................................................................. 31
   2.5.2 Bipartite graphs ............................................................................................................. 32
   2.5.3 Relation between bipartite and hypergraphs ................................................................. 32

## 3 FUNCTIONAL ARCHITECTURE recovery

3.1 Key concepts in functional architecture recovery ................................................................. 34
   3.1.1 recovery of product model for the embedded system .................................................... 34
   3.1.2 recovery of architecture model for the embedded software .......................................... 35
      3.1.2.1 Software model – implementation level ................................................................. 36
      3.1.2.2 Software model – design level ............................................................................... 37
      3.1.2.3 Alignment to AUTOSAR – standardisation ............................................................ 41
   3.1.3 Recovery of the functional architecture .......................................................................... 44
      3.1.3.1 Mapping .................................................................................................................. 44
      3.1.3.2 Recovery ................................................................................................................ 45
      3.1.3.3 Issues ...................................................................................................................... 46
3.2 Tool for functional architecture recovery ............................................................................. 47
   3.2.1 Sources for data collection ............................................................................................. 47
      3.2.1.1 Spectra ...................................................................................................................... 47
      3.2.1.2 Perforce .................................................................................................................. 47
      3.2.1.3 ESPRESSO repository ............................................................................................ 48
This introductory chapter presents the background and motivation behind this thesis and its purposes. The concept applied and the approach followed to fulfil the thesis purposes are also presented here. The last sections state delimitations and division of the work.

1.1 Background and motivation

The number of features included in new vehicle releases is always increasing. Consequently, a vehicle's electrical and computational complexity is also growing. The increased complexity makes it harder to keep a high standard of safety insurance and a good overview of a vehicle's architecture. Additionally, many aspects must be counted for in a vehicle's development process. It can be use cases, user functions, logical structure, modularity and flexibility to suit each unique customer. The development of embedded system in heavy trucks requires large teams, where communication and information sharing are critical.

Safety in vehicles is given more and more attention both by clients, governments and manufacturers. The ISO26262 standard deals with the safety aspects in vehicles and defines a work-flow and a set of design and development requirements to ensure high safety standard. Many vehicle manufacturers strive to comply with the ISO26262 standard by improving their work-flow and gaining a better overview of their vehicles' embedded systems architecture.

The ISO26262 safety standard requires that a vehicle's architecture is organised in a way that makes it possible to allocate different requirements to different parts of the system, without violating the set of safety constraints set by the manufacturer. Many engineering activities and information related to the standard shall also be generated. A high-level perspective of the architecture with traceability down to the operational level is therefore a need, which is usually solved by introducing a model of the system architecture. Having a model for the system architecture opens the door for future component reusing. Additionally, the traceability property makes it possible to develop mechanisms to check by logical relationships if a particular requirement, e.g. functional safety related, is fulfilled or not.

A problem that faces many manufacturers is that their vehicles are not produced according to well-defined models or architectures. Moving towards a top-down model-based development is a time consuming and costly process and may face objections in the company. A solution that is not equally obvious is to take a bottom-up approach and recover a model for already manufactured vehicles. The recovery process utilizes the vehicle's product data, such as software code and electronic schematics, to create an architecture representation.

Complex system models are generally not easily readable by humans, therefore the need for information filtering and generation of different views for different perspectives of a system.

SCANIA CV AB –called SCANIA from now on-, a leading truck and bus manufacturer wants to improve safety in their products and to be prepared for the ISO26262 standard. In addition the development of the trucks' embedded system at SCANIA does not follow a well-defined model. Thus, SCANIA has found a need to obtain a better overview their trucks' embedded system architecture. For these reasons, SCANIA R&D has hosted this thesis work as a part of a project called ESPRESSO.
1.2 Purpose

The main purpose of this thesis is to investigate possible ways to automatically recover the functional architecture (defined in section 1.6) of the embedded software in a truck. A second purpose is to figure out what data that needs to be stored in a new database, called ESPRESSO repository (defined in section 3.2.1.3), in order to support the auto-generation task. In other words, the following questions need to be answered:

1. Is it possible to automatically recover the functional architecture of a truck’s embedded software from the existing data infrastructure at SCANIA?
2. What data and data structure should exist in the ESPRESSO repository?

The work shall demonstrate the possibilities to automate tasks that are done manually today.

1.3 Solution concept

The work done in this thesis is based on a solution concept that was developed early in the thesis work. The concept defines a general work-flow to reach the goal of auto-generating the functional architecture of a truck's embedded software. Therefore, the concept equally fits the task of auto-generating other views or reports of the system.

Figure 1 visualises the concept and clearly defines three activities in the way to get the end product. The following list describes the detailed steps in the concept:

- **Data crawler**: in this activity data belonging to a particular truck's embedded system is collected from multiple data sources.
- **Model constructor**: is the activity that takes a product's definition data and uses it to create a structured model for the current truck's embedded system. The created model follows a meta model that defines its structure, which can be freely specified by the user. This is the step from unordered raw data defining the embedded system to an ordered model.
- **Analyser/View generator**: these activities take the product model as an input, and optionally an additional user input, in order to generate reports or views related to the truck's embedded system.

The following list defines the outputs from the above three activities:

- **Product's definition data** is the output of the first activity. This product is the collection of unordered raw data that defines the current truck's embedded system.
- **Product model** defines the embedded system, or part of it, in the current truck. The model can consist of multiple sub parts. In this thesis work it consists of two sub-models: one for the whole embedded system, and another for the embedded software.
- **Reports/views** are the end products that the user is meant to directly interact with and thus are human readable/viewable products.

The functional architecture is a view of the system and is an end product that is meant to be viewed by a user. Thus, the functional architecture generator, containing all necessary algorithms and logic, is a view generator. This means that by applying a different view generator or analyser, the end product can be totally changed according to the user's needs, which is a great flexibility and an opening for future work.
1.4 Approach

The work started with a background study to gain a proper understanding of the problem. The activity included identifying previous master thesis works, papers and other works in the area and determining the necessary knowledge to acquire in order to reach the goals. The areas of interest were meta-modelling, AUTOSAR\(^1\) and functional safety as defined in ISO26262. A major focus in the background study was to understand the existing infrastructure and data sources at SCANIA.

In parallel to the background research, simple implementations of the tool chain, described in figure 1 and called ESPRESSO demonstrator, started in order to gain better understanding of what is needed and required to achieve the thesis goals. Since the R&D department at SCANIA follows the agile method, the implementation work in this thesis was iterative until the final product was achieved. The implementation was internally presented in SCANIA at several points during the thesis work.

During the implementation of the ESPRESSO demonstrator, data that need to be contained in the ESPRESSO repository was determined along the way. At the end of the implementation, a first version of the repository was fully developed. In the same way, different parts of the truck’s embedded system architecture that need to be modelled were determined along the way. At the end of the implementation, a fully developed design model of a truck’s embedded software has been achieved; see figure 2 for a graphical overview of the work process.

\(^1\) AUTOSAR is a worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry. [http://www.autosar.org](http://www.autosar.org)
The first version of the ESPRESSO demonstrator could only show the Electrical Control Units (ECUs) existing in a truck that is chosen by the user. The second version resolved data dependencies between different ECUs over the CAN bus and created a view to represent that information. As a last development step, the third version could generate functional architecture view.

![Diagram](image.png)

Figure 2: A chart illustrating the process of the work - time flow from left to right

### 1.5 Delimitations

This thesis work has been limited to study and model the embedded system architecture in a SCANIA truck with focus on the embedded software. The embedded software that is modelled is only the highest layer in an ECU’s software which is a component-based layer. The recovery of the requirements on those software components with accordance to the ISO26262-8 is out of the scope of this thesis work.

The ECUs’ software architecture and CAN-bus signal information are parsed from each ECU’s source code and used in the ESPRESSO demonstrator. This work is not a part of this thesis and is assumed to be a ready input. In addition, the support of sensors and actuators as well as the end of line parameters are not covered in this thesis work.

### 1.6 Definitions

The following terms are used in this thesis and are described in the following subsections.

**Functional Architecture**

According to the ISO26262 standard an architecture is a “representation of the structure of the item or functions or systems or elements” [1, Sec. 1.3]. However, the functional architecture view that is proposed by this thesis is a representation of the architecture of the highest layer in an ECU’s software describing data dependencies between software components in the layer. Additionally, UFs are mapped to the software components.
1.7 Division of work

This thesis has been conducted by Ahmed Zamouche and Oussama Chammam. Both students were involved in the work presented in this document. However, in order to reach the goals and to meet the deadlines each student has invested more time in the following parts:

Ahmed has been mainly working on:
- ESPRESSO demonstrator software architecture and development
- Data mining from different data sources
- Background study of the functional allocation, and ISO26262
- Mapping ACs to UF s and recovery and visualisation of the functional architecture
- Formulation of the embedded system product model

Oussama has been mainly working on:
- Study on graph theory
- A background study of AUTOSAR
- Formulation of embedded software architecture model including abstraction of the CAN bus
- Mathematical expression of the embedded software architecture model
- Mapping the target system to AUTOSAR (alignment)

Additionally, the work has been carried out with a tight collaboration with the master students Oscar Molin, Behrooz Mokhtari and Juan Greco, which are all members of the ESPRESSO project. Oscar Molin has provided a representation for the ECUs' high-level software architecture by parsing ECUs' source code. Also by parsing ECUs' source code, Behrooz Mokhtari and Juan Greco provided CAN message information related to each ECU. The provided information by the two groups has been used as input to the ESPRESSO demonstrator.
This chapter starts by presenting a product abstraction model that is derived from the EAST_ADL2 standard and used in this thesis to classify the developed models. Next section describes the structure of the target embedded system.

Two sections in this chapter are dedicated for the widely used and supported automotive standards: ISO26262 and AUTOSAR, stating mainly parts that are related to the work done in this thesis. The relation of the ISO26262 standard to this thesis is obvious since it was one of the motivations behind this thesis. The AUTOSAR standard on the other hand was used as a reference and an inspiration source when developing architecture models for parts of the embedded system in a truck.

Last section presents two types of graphs which were used in this thesis to model the software architecture.

### 2.1 Abstraction levels as defined in EAST-ADL2

EAST_ADL2 is an architecture description language developed by the ATESSST project (www.atesst.org) for the automotive embedded system domain [2]. EAST_ADL2 defines a five levelled vehicle abstraction model; see figure 3. This thesis has used this model to describe the different abstraction levels in the developed embedded system models. Starting with the highest level, the five abstraction levels are [2]:

- **Vehicle level**: a top view of the vehicle's electrical/electronic system, describing it as a set of features.
- **Analysis level**: describing the vehicle as a set of logical functions (translating the features described in the level above).
- **Design level**: this level deals with the allocation of resources and partitioning of software.
- **Implementation level**: this level describes the software and hardware architecture of the vehicle. This level does not describe how the different software/hardware components are realized.
- **Operation level**: the realization of the software and hardware building units is described in this level, which among others includes source code files and hardware schematics.

---

Figure 3: Vehicle abstraction levels as defined in EAST-ADL2 architecture description language
2.2 Target system architecture

The structure of a product at SCANIA is hierarchical. This property was easy to observe from the data stored in Spectra database (more about Spectra in 3.2.1.1). A product consists of a list of main parts which are actually mounted on it. The main parts might consist of sub-part(s).

Inside a SCANIA truck there is a distributed system consisting of ECUs which are connected by several bus-systems. Figure 4 shows SCANIA embedded electrical system. The ECUs are listed as main parts. Every ECU entry in the product definition consists of a SW version and a HW version\(^2\).

2.2.1 Physical structure

The physical architecture, as it is illustrated in figure 4, consists of set of ECUs which are connected to a set of sensors, actuators and to one of the CAN-buses. There are two types of ECUs:

1. Physical ECU: is a physical unit which might contain one or several logical ECUs
2. Logical ECU: which is defined as a group of logical functions

An ECU can be on one of the following buses: RED, GREEN or YELLOW, with a unique address as defined in the address spaces at SCANIA and in accordance to the J1939 standard [3].

![Figure 4: SCANIA SESAMM architecture [41, p. 33]](image)

2.2.2 CAN communication

The ECUs exchange information using the Controller Area Network (CAN) - SAE-J1939 and ISO-11992 protocol-. Control units are located on different buses according to their criticality. The buses are bridged by special ECUs which have gateway functionality. The Coordinator System (COO) is the ECU that connects three main buses of the SCANIA Electrical System [4].

\(^2\) This is true only for the in-house ECUs
• Green bus: connects units that are least critical and mostly related to the user comfort, example FM radio
• Yellow bus: connects units that support the main functionality of the vehicle.
• Red bus: connects critical units, example the gear box management System (GBM).

Every ECU has a list of predefined messages that are transmitted and received. Every message has the following properties:
• Priority: value of 0 has the highest priority. This value is used to control the message priority
• Broadcast: determines if the message is transmitted as broadcast or not
• P2P: determines if the message is transmitted as Peer-to-Peer.
• DestAd: the destination ECU address, which has a meaning only if the message is transmitted as P2P
• PGN: Parameter Group Number refers to the value of the PDU format (PF), PDU specific (PS), the Data page bit and the reserved bit in the arbitration field of a CAN data Frame[5, p. 8]

In addition to these properties, a message carries signals. A signal has the following properties:
• Name: descriptive name of the signal
• BitLength : specifies the number of bits that the signal occupies in the message
• StartBit: specifies the start bit of the data, which is the least significant bit starting from the message data
• Unit: specify the physical unit of the signal

Figure 5: CAN Data Frame as Defined by J1939[5, p. 8]
2.2.3 Terminology and definitions

In the target system, global variables are managed by a module called Real Time DataBase. This module provides services for registering new variables and read/write access. Signals delivered/transmitted are stored in the RTDB.

Table 1 presents additional software related terms used at SCANIA, which will be helpful to understand the software architecture.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional architecture</strong></td>
<td>The SESAMM functional architecture consists of the complete set of defined user perceivable functions (so-called user functions). It defines what part of the physical architecture that is involved in the performance of each defined user function.</td>
</tr>
<tr>
<td><strong>User Function (UF)</strong></td>
<td>Is an electrical-related service as perceived by a user. The implementation of a user function can involve none, one or several ECU systems. In the first case it is referred to as a Discrete Electric Circuit (DEC).</td>
</tr>
<tr>
<td><strong>System Function</strong></td>
<td>Is local to a single ECU and is not necessarily perceivable to a user. E.g. a system function can be a temperature regulation algorithm or a process for evaluating one of several conditions. Normally, more than one system function is involved in the implementation of a complete user function</td>
</tr>
<tr>
<td><strong>Allocation Element (AE)</strong></td>
<td>An Allocation element is logical module that has a defined task and contribute to achieve one or more user function[6]</td>
</tr>
<tr>
<td><strong>Function Variable (FV)</strong></td>
<td>The function variable is an abstraction and labelling of information, used to trace the data flow of the information sent and received.[6]</td>
</tr>
<tr>
<td><strong>Functional allocation Description (FAD)</strong></td>
<td>The Functional Allocation Description (FAD) is a work product in term of document. The document describes the ECUs that implement a particular user function, the allocation of AE to ECUs and the flow of physical signals labelled as FV.</td>
</tr>
</tbody>
</table>

To clarify the concept and term usage an example is presented in figure 6 which shows Fuel Level Display user function. In this example, the truck has a liquid fuel tank and no Bodywork Communication Interface (BCI) ECU is installed. The AE Engine Information Provider provides the COO with FV FuelRate. The latter is transmitted inside the FuelEconomy CAN Message on the red bus. The AEs Fuel Sensor and Parking Brake, located internally in the COO, provide the AE Fuel Level Estimation with FuelLevelAnalogInput and ParkingBrakeHighApplied FVs respectively. The FuelLevel FV is transmitted to Instrument Cluster (ICL) ECU. In the case of low fuel level, the FV LowFuelwarning is broadcasted.
The amount of details that should be included in the FAD is not determined. Most of the AEs participate in the implementation of one or many UFs. This creates a web of interconnected AEs. At the creation of the UF diagram the creator has to make a decision on how many details to include, otherwise it could end-up with a global diagram [6, Ch. UF – User Function ].

### 2.2.4 Software structure

Due to the increase in software complexity and the number of user functions, the software is divided in many layers [7, Ch. 1]. Figure 7 shows the layered architecture of the software. The ECU hardware resides on the bottom and sensors/actuators are connected to the IOs. *Hardware abstraction layer* (HAL)\(^3\) abstracts the hardware and is usually shared among all ECUs. *Midd* is the second software layer which provides services such as the CAN communication, and memory access. Finally, the *application layer* is on the top of the architecture. This layer consists of smaller independent elements called *application components*, which implement different system functions as well as user functions.

---

\(^3\) Known as common platform at SCANIA

---
2.2.4.1 Application components

Application components (AC) are pieces of software in the application layer of the source code (Typically one file for each AC). Values that are contained in signals are stored in RTDB variables, and thus the only way to read/write to a signal is by using the write/read calls provided by the RTDB manager module. ACs exchange data usually through RTDB variables even in case the data is transmitted to another ECU since signals are read/written from/to RTDB variables [7]. However, there are exceptions where data is transferred as arguments in direct function calls when the ACs are located in the same ECU.

ACs can not directly communicate with network drivers, they are only aware of the Midd layer interface, which abstracts the network and offers a single system view.

2.3 ISO26262

The ISO26262 standard is a specialisation of the standard IEC 61508 for Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. Figure 8 shows an overview of the ISO26262. The standard consists of 10 parts. Parts 1-2 describe the process. Parts 3-7 specify a safety life cycle. Parts 8-10 discuss extra supporting processes, analyses and guideline [8, p. 3].

To start with, the item is defined as system or an array of systems to implement a function at the vehicle level [1, Sec. 1.69], where

- A system is a set of elements that relates at least a sensor, a controller and an actuator with one another [1, Sec. 1.129]

and

- An Element is system or part of a system including components, hardware, software, hardware parts, software units [1, Sec. 1.32].

In addition, the item boundaries, the hazard and risk assessment are all work products of part 3.

Part 4 initiate the product development at the system level and the specification of the technical safety requirements followed by the system design. The technical safety requirements are then refined and specified to each HW and SW subsystem in part 5 and 6 receptively. Those requirements are then mapped to an architectural element.

The process of mapping the safety requirements needs a good knowledge of the architecture. The ISO26262 define the architecture as “representation of the structure of the item or functions or systems or elements” [1, Sec. 1.3].

---

4 For more details on the Allocation to HW/SW see [9, Sec. 7.4.5]
This master thesis work attempts to recover the functional architecture of the embedded system. The ISO26262 standard assumes a top down approach where the functional architecture is defined before detailed technical software architecture. On the other hand, for existing products, recovery of the architecture would play an essential role. Atomic units of software obtained from source code parsing will constitute the building blocks of the proposed architecture. Those units are called Application Components ACs in this thesis which is the equivalent to the Software Component term used in the ISO26262 standard. To be qualified as a software component, the ISO26262-8 requires that the component [10, Sec. 12.4.2]:

1. Complies with the requirement of the software component as described in ISO26262-8 clause 12.4.3.1 [10, Sec. 12.4.3.1]

2. Provide evidences the component comply with the requirements. ISO26262-8 clause 12.4.3.2 [10, Sec. 12.4.3.2]

3. The structural coverage shall be measured to evaluate the completeness of the test cases ISO26262-8 clause 12.4.3.3 [10, Sec. 12.4.3.3].

4. The evidence of compliance is only valid for an unchanged implementation of the software component ISO26262-8 clause 12.4.3.4 [10, Sec. 12.4.3.4]
2.4 AUTOSAR

AUTomotive Open System Architecture (AUTOSAR) is a de facto standard for automotive software system development. It is a cooperation between vehicle, electronics, semiconductor and software tools companies [11].

The main goal of AUTOSAR is to set a standard for basic functions and interfaces in the embedded software. It also aims to make software parts more independent of the underlying hardware and more movable between the different ECUs in a vehicle. Part of AUTOSAR's aim is to simplify maintenance work and upgrades of a vehicle's embedded software during its life cycle [12] [11].

An essential idea behind AUTOSAR is to make Software Components (SW-C) re-usability possible and to standardise their interfaces so that they are independent of the context including the underlying hardware [11]. The system shall be more flexible and easily scalable since adding, removing and/or relocating SW-Cs does not require any modification. This opens the possibility to reuse trusted and well-tested SW-Cs which should result in more reliable systems. The goal is to reach the point where it becomes possible to mix hardware and software solutions from different manufacturers seamlessly. The focus here is to provide an infrastructure that mainly facilitates the optimisation of SW-C allocation and not the performance of the SW-Cs themselves [12].

The goals and the approach of the AUTOSAR standard are so general that it can be partially applied to other domains than the automotive, with just small modifications.

The AUTOSAR standard handles three topics: software architecture, application interfaces and methodology. The topics are briefly covered in the following sections.

2.4.1 Software architecture

The AUTOSAR standard defines three main software layers with different purposes and clearly defined interfaces [13].

2.4.1.1 Basic software layer

Basic Software (BSW) layer is the bottom layer that is hardware dependent. This layer delivers basic functionalities and services to the layers above it. This layer contains OS functionalities, like scheduler, and is layered itself; read in [13], [14] and [15] for more details about the BSW layer.

2.4.1.2 RTE

Run Time Environment (RTE) layer implements the system services that provide the communication possibilities between the SW-Cs as well as scheduling of those [16]. This layer is located above the BSW layer and is tightly coupled to it, among others because of a common task scheduler used by both layers. While the RTE layer is dependent on both the BSW and the SW-C allocation, it totally abstracts any hardware and software dependency for the above layer.

The RTE layer implements communication channels between the SW-Cs, so-called Virtual Function Bus (VFB) [17] and the interaction between SW-Cs and the OS. The VFB abstracts any hardware dependencies so that a SW-C can not see whether another SW-C is residing on the same ECU or in a remote ECU. The AUTOSAR standard allows exceptions where a SW-C bypasses the RTE and directly communicates with the BSW or with the hardware whenever it is necessary to achieve a special functionality.
2.4.1.3 Application layer

The AUTOSAR application layer is a component based layer for the application programs. It consists of SW-Cs that realises desired functions for the vehicle user. All the re-usability that the AUTOSAR standard aims for is found in this layer. Both the RTE and the BSW layers are designed to support SW-C re-usability by totally isolating the SW-Cs from the hardware so they only see and utilise the VFB [13] – it is therefore not possible for a SW-C to know its allocation information.

The re-usability is also achieved by defining a strict interface for the SW-C that the SW designer must follow, with some exceptions. The SW-C interface is the only restriction for SW-C designers, while the internal design is meant to be the area where different manufacturers compete for best performance and functionality.

Development of SW-C

Developing a SW-C deals mainly with three aspects [17]:

- Design the component model (ports, interfaces, component type etcetera)
- Build an internal behaviour file (the code defining the component behaviour)
- A description of the SW-C implementation, which sets requirements on the RTE (e.g. what events it will receive through RTE and how often it shall be called)

SW-C types

The choice of SW-C type depends on what it shall perform; the component types defined by AUTOSAR are presented in the following list [17]:

- Application SW-C which is an atomic SW-C realising a complete or a part of a functionality.
- A composed SW-C which is a SW-C containing other atomic or composed SW-Cs. The contained SW-Cs can be connected together building on internal network (which is a part of the VFB).
- Sensor-actuator SW-C is an atomic SW-C that provides other SW-Cs with the possibility to access a sensor/actuator.
- Calibration parameter component is an atomic SW-C that only provides other SW-Cs with calibration data.
- Service component provides services to other SW-Cs.
- ECU abstraction component is used for special cases where it directly interacts with the BSW layers and provide an abstracted interface for other SW-Cs.
- Complex device driver component is slightly different from the ECU abstraction component and is often used to serve critical resource demanding SW-Cs.

Ports and interfaces

The AUTOSAR standard defines two types of SW-C ports: a data requiring port (RPORT) and a data providing port (PPORT) [17]. Each port requires/delivers data in one of six possible ways, called port interfaces; see table 2 for the AUTOSAR graphical representation. A description of the six interfaces is presented in the following list:

- Client-server requiring that a client SW-C sends a request to a server SW-C in order to receive data. The standard allows the client to have arguments in the request so that the
server performs its task. When the server handled the request, it replies with either a bad or a good response.

The AUTOSAR standard only allows the connection of \( n \) clients to a single server. In other words, an RPORT (a client) is only allowed to be connected to a single PPORT (a server), while the PPORT can be connected to an arbitrary number \( n \) of RPORTs.

- **Sender-receiver** meaning that data sent by a sender is not triggered by a receiver. Except from the optional data delivery acknowledgement, the sender and the receiver components are decoupled in terms of synchronisation. The sender-receiver interface allows both multiple senders to send data to a single receiver (\( n:1 \)) and a single sender to send to multiple receivers (\( 1:n \)), but not multiple senders to send to multiple receivers (\( n:m \)).

The data can be sent in one of two modes: either *last-is-best* (only in case of \( 1:n \) configuration), or in *queued* mode (possible for both \( 1:n \) and \( n:1 \) configurations). Last-is-best means that the last sent data by the single sender is the only valid data. Queued means that all the data sent by the sender(s) are valid and therefore should be kept in the receiver’s queue as long as the receiver did not read it yet.

- **Parameter interface**, which is found in calibration SW-C and is used by one or multiple SW-Cs to calibrate their internal logic.

- **Non-volatile data interface** is used to give other SW-Cs the possibility to access non-volatile data.

- **Trigger interface** is used to externally trig SW-Cs’ execution.

- **Mode switch interface** is used to notify SW-Cs about changed modes.

To connect SW-Cs with each other through the VFB, a so-called assembly-connector is used.

Table 2: AUTOSAR graphical symbols for the defined port interfaces – source for the symbols: [17]

<table>
<thead>
<tr>
<th>Port interfaces</th>
<th>Port type</th>
<th>RPORT</th>
<th>PPORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sender-receiver</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-volatile data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-29-
The internal behaviour of an application SW-C is defined by its implementation files (code files) and are divided into a so-called runnable entities, or just runnables [17]. Runnables are the smallest logical elements in a SW-C, which correspond to pieces of codes with freely chosen lengths. One or multiple runnables are encapsulated into so-called tasks which are scheduled by the RTE according to the SW-C implementation description.

Runnables can be one of two types, either blocking or non-blocking. A blocking runnable contains code that waits for non-deterministic behaviour or input in order to finish execution, and the opposite for non-blocking runnables.

Implementation description

To document and describe a SW-C’s model and properties and link it to code files, AUTOSAR defines its own XML template, ARXML. This XML format is used as input to AUTOSAR compatible tools that generate RTE and BSW files. The template is described in details in [18]. Appendix C presents an example SW-C described in ARXML.

2.4.2 Application interfaces

To make the exchange of SW-Cs from different designers possible, AUTOSAR defines application SW-C interfaces for the common applications found in a vehicle, including the interfaces toward the VFB. Depending on the application's domain, those interfaces fall into five categories. If a particular application falls outside the defined interfaces, the designer is allowed to design his own interface; otherwise he must follow the AUTOSAR interface specification [19].

2.4.3 AUTOSAR methodology

AUTOSAR standard methodology defines needed steps to generate an ECU executable, divided into four main phases [20]:

- Configure system
- Extract ECU specific information
- Configure ECU
- Generate executable

The methodology is not a complete work-flow and does not tell the order in which the steps should be taken, but only presents dependencies between those.

A major part of the methodology consists of presenting templates that shall be followed to describe different parts of the system, configuration files and the AUTOSAR meta model.
2.5 Graph theory

Graphs are a very useful tool to view and model systems, architectures and organised elements in a mathematical way. Graphs have been used in various areas since many decades and become a natural way to express biological structures and computer software’s architecture, as examples. Graph theory is the part of mathematics that expresses graphs in numbers and matrices and develops algorithms and operations around those. The following sections present two types of graphs that are used in this thesis work.

2.5.1 Hypergraphs

A hypergraph, like a standard graph, has a set of vertices connected with edges. The only difference is that the edges are no longer restricted to only connect two vertices [21]. A single hypergraph can contain multiple edges, each with a different number of endpoints. Like standard graphs, hypergraphs can also be directed and thus an edge can connect an arbitrary number of source vertices and an arbitrary number of target vertices; see figure 9.

A hypergraph H is represented with a set of vertices \( V \) and a set of so-called hyperedges \( E \). The graph can also be represented with a \((0,1)\)-matrix, so-called incidence matrix [22, p. 2], where rows represent vertices and columns represent hyperedges. A value 1 in position \((i,j)\) means that vertex \( v_i \) is connected to edge \( e_j \), while a zero means no connection. This representation has the advantage of allowing the connection of more than two vertices to the same edge compared to the adjacency matrix for a standard graph [23].

When representing directed hypergraphs with an incidence matrix there is a need to encode the direction of the connection between a directed hyperedge, called a hyperarc [22, p. 3], and a vertex. A \((0,1)\)-matrix is therefore not enough. Instead, a \((-1,0,1)\)-matrix is used where -1 encodes a source vertex that is connected to a hyperarc, and 1 encodes a connected target vertex [22, p. 2]. Figure 10 shows the incidence matrix corresponding to the hypergraph in figure 9.
2.5.2 Bipartite graphs

A bipartite graph is a two-coloured graph whose vertices can be divided into exactly two disjoint sets [24, p. 144], called partite sets [25] or bipartitions [26]. A vertex in any of the two bipartitions is non-adjacent to any other vertices in the same bipartition [26]. Thus, edges in a bipartite graph always cross both bipartitions' boundaries and is never internal in a single bipartition; see figure 11.

A bipartite graph $G$ is mathematically represented with two bipartitions $U$ and $V$ and a set $E$ containing all edges in the graph. It is also practical to represent a bipartite graph with a single matrix $B$, a so-called biadjacency matrix. The matrix $B$ has as many rows as there are vertices in the bipartition $U$ so that row $i$ represents all edges connected to the vertex $u_i$. In the same way a column $j$ of the matrix $B$ represents all edges connected to the vertex $v_j$ in the bipartition $V$. Thus, matrix $B$ has the dimension $\text{dim}(U) \times \text{dim}(V)$ where $b_{ij}$ has a value 1 when an edge exists between vertex $u_i$ and vertex $v_j$, otherwise 0. Generally, $b_{ij}$ can also have other values in case of multiedges or weighted edges. In the same way as for directed hypergraphs, $b_{ij}$ can have the value -1 if the graph is directed, meaning that $u_i$ is a source vertex. Figure 12 shows the biadjacency matrix for the bipartite graph shown in figure 11.

![Figure 11: An example bipartition graph](image)

![Figure 12: Incidence matrix for a bipartite graph](image)

2.5.3 Relation between bipartite and hypergraphs

In some applications it is easier to handle/display a particular type of graphs than other types, thus mapping between different graph types can be useful.

Figure 13 shows graphically an easy way to map hypergraphs to bipartite graphs. Given a hypergraph $H = (V,E)$ and a bipartite graph $G = (U_b,V_b,E_b)$ where $V$, $U_b$, and $V_b$ are sets of vertices and $E$ and $E_b$ are sets of edges, the mapping can mathematically be expressed as: the vertices in the bipartition $V_b$ maps to all hypergraph's vertices, and the vertices in the bipartition $U_b$ map to the hyperarc. The set $E_b$ represents sub-edges connecting a hypergraph vertex with a hyperarc.

From the above mapping, the biadjacency matrix maps exactly to the hypergraph’s incidence matrix which is a very useful property from a computer science’s perspective, since it requires no matrix operations, just interpreting the matrix differently.

-32-
Figure 13: An example of mapping from a hypergraph to a bipartite graph
3 FUNCTIONAL ARCHITECTURE RECOVERY

Following the goal and approach described in Chapter 1 and the technological content given by Chapter 2, the functional architecture for a user selectable UF could be recovered after designing and implementing the following steps:

- Analysing and consolidating related product information.
- Recreating a product model for the embedded system (operational level)
- Recreating an architecture implementation model for the embedded software (implementation level)
- Recreating an architecture design model for the embedded software (design level)
- Recreating a functional architecture (design level)

Observe that the architecture implementation model and the architecture design model are described earlier in this report as a single model called architecture model since both models capture the structure of the software but on two different levels. The main difference is on the level of hardware abstraction these models imply. Also, the architecture design model covers all ECUs in a truck while the architecture implementation model only covers a single ECU.

All the above recreated models are not part of SCANIA's goal, but are necessary to pass in order to reach the final goal: recreating a functional architecture. Again, the difference between the models is on the level of hardware abstraction.

To automate the above activities, a tool chain prototype called ESPRESSO demonstrator has been built. The tool allows creation of these models as well as the generation of appropriate views. In particular, an alignment of the proposed embedded software architecture models to AUTOSAR standard has been proposed for standardization purposes.

It should be highlighted that the tool doesn't provide a fully automated solution with the current data structure at SCANIA. Instead, a new database called ESPRESSO is proposed to support the automated task. Additionally, the data structure at SCANIA should be made consistent. Therefore, and despite the purpose of this work, the above listed steps included some manual work.

The following sections describe the key concepts and their implementations.

3.1 Key concepts in functional architecture recovery

3.1.1 recovery of product model for the embedded system

The embedded system product model consists of a set of ECU modules, where every ECU module consists of:

- A software architecture model consisting of the following objects:
  - A list of all ACs present in the ECU.
  - A list of RTDB variables used to exchange data between ACs
  - A list of edges specifying if an AC is reading/writing from/to and RTDB variable
- A list of messages and the signals that the ECU receive/transmit

Figure 14 illustrates the model. The arrows indicate the read or a write operations performed.
Since every signal is connected to an RTDB variable, the relation between the signals and the ACs can be deduced. The embedded system structure is stored in an XML file; see Appendix B for DTD.

### 3.1.2 recovery of architecture model for the embedded software

In this section a model for the application layer in a truck's ECUs is introduced. The lower level layers of the software including the midd layer are not included in the model. The reason for limiting the model is SCANIA’s interest in the component based application layer (see 2.2.4 for details about the layer).

Data flow graphs (DFG) are widely used in computer science to model flow of data through software architecture [27], [28], [29]. Since DFG implies knowledge about data related time properties, it usually requires a simulation to determine different data paths along with the time. The approach taken in this thesis to model software is not simulation based, but only looks at data dependencies between ACs independent of time. Therefore, data dependency graphs (DDG) have been used instead which is a form of DFG without time property [30] [31].

Because of the complicated nature of the tasks ECU software performs, the DDG contains loops and thus is not a tree graph.
The following two sections will describe the steps taken to create such a model over a whole truck, together with the challenges met and how they have been solved.

### 3.1.2.1 Software model – implementation level

Taking in consideration the introduced delimitation, a single ECU software can be modelled with a small DDG of application components, abstracting all other underlying details; see figure 15A.

It is fairly easy to see that the architecture maps exactly to a directed *hyper-data dependency graph* where hyperarcs represent data-dependencies; see figure 15B. The disadvantage with this type of graphs is that it is hard to find support for it in algorithm libraries, and that the algorithms applicable on it are a small subset of all existing graph algorithms. Additionally, most of the graph related research handles only standard graphs.

An alternative is to convert the hyper-DDG to a directed bipartite DDG. One bipartition will then represent application components, and the other will represent connections (hyperarcs) between the applications. A second look at ECU software architecture in SCANIA gives that data-flow between application components is *mainly* performed through shared variables, locally to each ECU. Thus, there exist some exceptions where data flows between application components through direct calls, bypassing any shared variables and making it harder to detect when parsing the software. Therefore, to simplify software model creation and detection of data-flow, the following assumption has been made:

**Assumption 1:** “Application components are only allowed to communicate through shared variables, excluding the possibility for direct communication or any other ways”

Given the above assumption, the shared variables correspond to the hyperarcs in the hyper-DDG and correspond to the vertices in one of the bipartitions in the bipartite DDG; see figure 16.

The above approach assumes that the ECU’s application layer architecture is known. In the ESPRESSO project, the architecture is recovered by parsing ECU source code. The source code parser was developed by Oscar Molin where its output was used as input for this thesis work. If an ECU was developed outside SCANIA, then the source code is not available for parsing. In that case, the ECU is modelled as a single application component, corresponding to a single vertex in the DDG.

![Diagram showing data dependency between application components and its mapping to a hyper-DDG](image)
Data dependency between application components

Figure 16: Mapping from a SCANIA application component architecture to both a hyper-DDG and a bipartite DDG, showing that shared variables map to hyperarcs and to vertices respectively

3.1.2.2 Software model – design level

The previous section described a straightforward approach to model a single ECU’s software. This section builds upon the previous one and presents an approach to unify all ECU-DDGs, which will result in a single DDG for the whole truck’s embedded software.

By unifying ECU-DDGs, it is meant to connect application components located in different ECUs using information about data-flow over the CAN bus. By looking at all possible CAN messages’ properties and how they are received in the different ECUs, it is possible to connect all ECU-DDGs together. The CAN messages are the deliverers of data between application components located in different ECUs. By connecting the DDGs together, three new issues arise; the following sections describe and propose solutions for the issues.

Issue 1 – abstracting communication software components resulting in direct link between shared variables

This issue arises from the abstraction of software components handling transmission and reception of CAN messages in an ECU. The result is that the link between two ECUs (over the CAN) becomes a direct link between two shared variables, and clearly data can not flow between two data holders (shared variables) with no involvement of any application component (a runnable software); see figure 17.
The chosen solution was, in the case of a hyper-DDG mapping, to merge all connected hyperarcs, and in the case of a bipartite DDG mapping, to merge all communicating vertices that are of the same bipartition type (type is either a shared variable or an application component); see figure 18.

Figure 17: Issue 1 as a result of the abstraction of the CAN bus and the transmitter/receiver components.


**Issue 2 – Chain communication over CAN in different directions resulting in false data-flow path**

Figure 19 shows an application component network that cannot be modelled with the developed model. In the case of modelling the network with hyper-DDGs, it would result in two communicating hyperarcs, and modelling it with a bipartite DDG would result in two connected vertices from the same bipartition type (shared variable type).

Merging the hyperarcs or the vertices in the bipartite DDG, as suggested in the previous section, would mean that data flows from vertex AC7 to vertex AC2, which is not true; see figure 19.

The problem arises from chain communication of shared variables in two different data-flow directions. By chain communication it is meant that multiple shared variables are connected through CAN bus communication. By different data-flow direction it is meant that data is not forwarded between the shared variables, instead at least two shared variables in the chain are writing to the same shared variable. Chain communication of shared variables in the same data-flow direction would not introduce any false data-flow paths because the data is just forwarded from one shared variable to another; see figure 20 for an example.

This has led to the following assumption:

**Assumption 2:** “No shared variables are chain communicating over the CAN bus with different data-flow directions”
Figure 19: Example of merging of shared variables that communicates in different data-flow directions, which introduces a false data-flow path

Figure 20: Example of merging shared variables that communicate in the same data-flow direction - no false data-flow paths are introduced

**Issue 3 – Writing to a shared variable both from internal and external ACs**

Figure 21 shows how merging two shared variables in the DDG introduces a false data path from AC4 to AC1. The reason for the incorrectness is that the shared variable V3 is written to both internally (by AC4) and externally (from ECU1), and data in V3 is not read by ECU1. This means that data does not flow from V3 to V2 (ECU1) and thus merging V3 and V2 will hide this single direction property of data-flow.
This problem has not been directly solved, but led instead to the following assumption:

**Assumption 3:** “A shared variable is not written to both internally and externally, except from the case where the data is written back to all external sources”

![Diagram of merging two shared variables](image)

**3.1.2.3 Alignment to AUTOSAR – standardisation**

In addition to the functional architecture view, mapping the embedded software model to AUTOSAR is also a view generation activity, which is the last activity in the concept. The standardisation is a second example of the wide possibilities available to automate tasks that otherwise maybe time consuming to do manually.

In this section two proposed ways of mapping the current SCANIA architecture to the AUTOSAR architecture are introduced. The difference between the two proposed mappings lies in the different interpretations of the shared variables. Since the embedded software model represents only the high-level application layer in a SCANIA truck's embedded software, then the model will be mapped to the Application layer in the AUTOSAR standard; the two lower layers, RTE and BSW, are not of interest in this context.

Note that the mapping and analysis done in this section concerns only topological aspects and does not cover the mapping with respect to the model behaviour.

**Approach 1 – A bipartite based approach**

Taking the bipartite DDG as a model, a SCANIA application component maps well to an AUTOSAR SW-C, and in case it consists of multiple sub-components then it maps well to an AUTOSAR composed SW-C.

Mapping the shared variables to the atomic AUTOSAR application SW-Cs, the communication between the AUTOSAR SW-Cs will always be 1:1, exactly like edges in a bipartite DDG. Since the AUTOSAR SW-Cs representing the shared variables does not correspond to SCANIA application components, they can be flagged with a special attribute so that the tool that takes the AUTOSAR representation as input is aware of the different SW-C behaviour.

The port interfaces have been chosen to be sender-receiver since SCANIA does not utilise any server equivalent mechanism when accessing a shared variable, read under 2.2.4 for more details. With the chosen interface, it is possible to model both read and write access to shared variables. Since the SW-Cs are 1:1 connected there is only one sender that sends to a single receiver. Additionally, since either the receiver or the sender must be a shared variable, the data should be received as last-is-best with no queue buffers in the receiver side. Shared variable access (edges in the bipartite graph) has been mapped to AUTOSAR assembly-connectors.
If an $n \times m$ matrix $B$ is the biadjacency matrix for the bipartite DDG representing the SCANIA software, then it is known that the number of vertices in the graph is $(n+m)$ which corresponds to the number of SW-Cs to be created in the AUTOSAR architecture description. The number of bipartite edges, and thus the AUTOSAR assembly-connectors, can be expressed as shown in formula 1.

$$\sum_{i=1}^{n} \sum_{j=1}^{m} b_{ij}$$

(1)

Table 3 summarises the mapping which is straightforward. Additionally, figure 22 presents an example of a bipartite DDG to AUTOSAR mapping.

Table 3: Mapping from bipartite DDG to AUTOSAR architecture

<table>
<thead>
<tr>
<th>Bipartite DDG element</th>
<th>AUTOSAR element</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a vertex from one set)</td>
<td>(an application SW-C)</td>
</tr>
<tr>
<td>(a vertex from the other set)</td>
<td>(an application SW-C)</td>
</tr>
<tr>
<td>(an edge)</td>
<td>(an assembly-connector)</td>
</tr>
</tbody>
</table>

Figure 22: An example of mapping of a bipartite DDG representing SCANIA application components, to AUTOSAR architecture

**Approach 2 – A hypergraph based approach**

Taking the hyper-DDG as a model, still a SCANIA application component maps well to an AUTOSAR application/composed SW-C. In a hyper-DDG, shared variables are not modelled as vertices, but as hyperarcs instead.

A hyperarc can be a link between an arbitrary number $n$ of senders (data-flow out from vertices) and an arbitrary number of receivers $m$ (data-flow in to vertices), which is a configuration that is not supported by the AUTOSAR defined interfaces (AUTOSAR only allows $1:n$ or $n:1$ configurations).

To map the hyperarcs to AUTOSAR architecture, each hyperarc is split into multiple $1:n$ (1 sender, $n$ receivers) port combinations resulting in expansion of number of receiving ports (RPORT) compared to the hyper-DDG; see figure 23. On the other hand, since the shared variables are modelled with edges (AUTOSAR assembly-connectors) this approach creates fewer AUTOSAR SW-Cs compared to Approach 1. If a hyperarc has $n$ sources and $m$ targets ($n:m$) then the total number of connected ports after splitting can be calculated with formula 2.

$$n \times (m+1)$$

(2)
Figure 23: Splitting a hyperarc increases the number of receiving ports in the AUTOSAR architecture description

Remember that the hyperarcs represent data dependencies and not data flows, therefore the split of these hyperarcs still preserve the information about the dependencies.

Since receiver components do not ask the sender for data or send any arguments to it, but just reads the value when it needs it, the port interface should be sender-receiver, which makes it possible to model both read and write access to shared variables. There is only one sender that is supposed to send a single value, representing the value stored in the shared variable, to multiple receivers. The data should be therefore received as last-is-best with no queue buffers in the receiver side.

The tool that is expecting this AUTOSAR description as input should be aware of the hyperarc split and convert them back to the original joined connections. That is, creating a single shared variable for each joined hyperarc. To make the merging possible, all AUTOSAR connectors that were joined in the original hyper-DDG representation shall have the same value of the customized parameter hyperarc-name, containing the name of the hyperarc they all together represent.

If an \( nxm \) matrix \( H \) is the incidence matrix for the hyper- DDG representing SCANIA embedded software in a particular truck, then it is known that the number of vertices in the graph is \( n \) which corresponds to the number of SW-Cs to be created in the AUTOSAR architecture description. The total number of hyperarcs in the hyper- DDG is \( m \), but the total number of AUTOSAR assembly-connectors becomes larger; see formula 3. The difference is due to the hyperarc split performed in order to map to AUTOSAR; see APPENDIX A for a complete formula proof.

\[
\sum_{j=1}^{m} \left( \frac{|h_{ij}| - h_{ij}}{2} \right) \times \left( \frac{|h_{ij}| + h_{ij}}{2} + 1 \right)
\]

(3)

Unlike the mapping schema in Approach 1, there is a need for preprocessing (splitting of hyperarcs) before a straightforward mapping is possible. Table 4 summarises the proposed mapping. Figure 24 shows an example of mapping a hyper- DDG to AUTOSAR.
Table 4: Mapping from hyper-DDG to AUTOSAR architecture

<table>
<thead>
<tr>
<th>Hyper-DDG element</th>
<th>AUTOSAR element</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a vertex)</td>
<td>(an application SW-C)</td>
</tr>
<tr>
<td>(a simple edge)</td>
<td>(an assembly-connector)</td>
</tr>
<tr>
<td>(a hyperarc)</td>
<td>(multiple assembly-connectors)</td>
</tr>
</tbody>
</table>

Figure 24: An example of mapping of a hyper-DDG representing SCANIA application components, to AUTOSAR architecture

3.1.3 Recovery of the functional architecture

This section describes how to auto generate functional architecture using the embedded software architecture design model, which was discussed in 3.1.2.

3.1.3.1 Mapping

There are around 470 UF types at SCANIA, where around 80 of them were realized in one of the trucks that have been studied in this work. The first step is to map those UF types to ACs. This step was done manually for every different software version since application component names and folder structure of the source code changed from one version to another. The mapping is stored in XML file in the ESPRESSO repository in the format described by Text 1. In contrast to the actual implementation where a UF is allocated to one or multiple AEs, a one to one mapping is adopted instead. The motivation behind it is a proposal by the ESPRESSO project to simplify responsibility distribution among software development teams.

<UF id="1860966" code="18" ca="1766669" ac="vsfc">Fuel Level Warning</UF>

Text 1: A UF entry in the mapping file

id: unique identifier, code: user function unique code,
ca: user function category, ac: application component name
3.1.3.2 Recovery

The recovery process takes the mapping of UF to AC and the software architecture design model as an input; see figure 25. The output, which is called Functional Architecture, shows data dependencies between ACs that are involved in the implementation of a particular UF that is chosen by the user.

The recovery process starts by the AC that is responsible for implementing of the UF. Figure 26 shows a graphical representation of the functional architecture. The AC responsible for implementing the UF is at the top of the tree -the root element-. The next level, is the set of the AC that reads shared variable or signals written to by the root AC. The coloured edges indicate that the transition is done over the corresponding CAN bus and that data flows from one ECU to another.

Figure 26: An illustration for a general functional architecture
### 3.1.3.3 Issues

The processes of recovering functional architecture in 3.1.3.2 needs to take into consideration the following issues

- **Loops**: they occur in
  - Scenario 1: An AC, reads and writes to the same variable/signal \( v_j \).
  - Scenario 2: An AC, in the functional architecture tree writes to a variable/signal \( v_z \) which is read again by an AC, that preceded AC, in the tree.

- **Details**: the amount of details that should be included in the functional architecture has to be determined
  - The depth of the tree has to be limited in order to be viewable and provide a meaningful tree which describe the user function properly
  - Irrelevant branches: not all branches in the tree are relevant/needed to get an overview of the current UF implementation and communication

The first issue can be solved by breaking the loop and only capturing data dependencies in one direction of the tree. The second issue can be solved by giving the viewer the possibility to limit the coverage of the tree to a selectable depth and the possibility to hide irrelevant branches. Another alternative is to store additional information in the ESPRESSO database that tells what branches are irrelevant for a specific UF to automatically eliminate them.

Figure 27 shows a real example of the functional architecture associated with the UF18, with one depth level of the tree. The \textit{vsfc} AC broadcast the \textit{EngineFuelLevel} signal over the CAN bus to the fuel AC located in the COO. The latter is responsible for estimating the \textit{FuelLevel} and \textit{LowFuelLevelWarning} if applicable, then broadcasting them to the ICL.

![Figure 27: Functional architecture of the UF18](image)

The \textit{vsfc} AC located in the EMS broadcast other signals which do not implement UF18. In figure 27 those branches are represented with crossed arrows.
3.2 Tool for functional architecture recovery

The ESPRESSO demonstrator is the tool that has been developed as a proof of concept and demonstrated the possibility to recover the functional architecture for a user selectable UF. The following sections describe how a truck's data was collected from different sources and how the demonstrated tool was designed.

3.2.1 Sources for data collection

A truck's embedded system model shall be based on the data describing the truck. The data shall be as close as possible to the reality; in other words to the data used in production. In this project, mainly three sources have been used to derive a truck's model. The following sections describe the different data sources and the data collection and extraction challenges associated with them.

3.2.1.1 Spectra

Spectra is a main frame application for handling conditional and time dependent product structure at SCANIA. Spectra can be accessed using Aros Work Station (AWS) [32] terminal emulator. The data are stored in hierarchical way by a database management system called Rosam[33]. The database holds information that defines products manufactured at SCANIA. Both the database and the mainframe application are called Spectra.

The database is hierarchically organised to easily describe the structure and hierarchy of the truck design, both hardware parts and logical elements (e.g. user functions realised by a particular component).

The original idea was to access the database through the Aros Transaction server (ATS) to be able to query the database in an SQL style. Since our request was denied, it was not possible to access ATS to automate the communication with Spectra. To solve the issue, the data has been manually copied from Spectra for each involved truck and used to create a database to simulate access to Spectra.

It was not possible to fully automate the data obtained from Spectra. The data were not always consistent. As an example, the information about software version for an ECU was stated in different ways depending on which truck it belongs to and who has entered the data. The main reason for the inconsistency was because the data were entered manually.

3.2.1.2 Perforce

Perforce is a Version Management System used by SCANIA for version control of the source code. Perforce provides many tools to access the server. The tool p4ftp have been chosen, that is an ftp server once installed on the local machine gives access to Perforce server using the ftp protocol [34].

The file structure in Perforce server is different from one ECU to another, and sometimes from ECU version to another version. Some source files are also kept outside Perforce system, which makes it not possible to fully automate source code fetching. To get access to source codes that are outside Perforce, an e-mail has to be sent with a request for access rights for each folder with a long time of waiting. To temporary solve the issue for a particular truck, part of the source files were locally stored.
3.2.1.3 ESPRESSO repository

The ESPRESSO repository is a database that is proposed for ESPRESSO project to contain data that is currently not found in any database at SCANIA in a machine readable format. The repository has been filled with data during the development phase. Information regarding the list of ECUs, their names, addresses, location on the bus and path to the source code were compiled from SCANIA archived documents and stored in the database.

3.2.2 Design of the tool

The ESPRESSO demonstrator tool consists of a software application written in C# using Visual studio 2012 development environment. Other programming languages such as HTML, JavaScript, CSS and XML were also used for user interface and persistent data storage. The following sections describe the different aspects of the ESPRESSO demonstrator tool.

3.2.2.1 Layered software architecture

In the design phase of the application the following aspects were taken into consideration: databases sources, intermediate meta model, and user interface. The implementation uses the 3-tier architecture [35] with the following layers: Presentation-, Business Logic-, and Data access-layer; see figure 28. The layers are described in the following sections.

Figure 28: The ESPRESSO demonstrator tool software architecture
**Presentation layer**

The Presentation is the top layer; it takes user inputs and displays the corresponding output. Figure 29 shows a screen-shot of the application main user interface. This layer has the following input and outputs

- **Input:**
  - Input field for the VIN and/or a signal name in the format *Vehicle: VIN [CANs: "signal name"]*
  - Two radio buttons to choose between two views:
    - SESSAMM architecture with data directions for the chosen signal
    - Functional architecture diagram

- **Outputs:** the application has the following outputs
  - Production date for the specified vehicle
  - SESSAMM architecture with available ECU
  - Signal information if the signal has been specified in the search field
  - Information about the message that the signal belongs to
  - ECUs that send or receive the signal are highlighted and arrows indicate data dependencies.
  - Functional architecture

![Figure 29: The ESPRESSO demonstrator screen-shot](image)
**Business logic layer**

This is the core of the application; it implements the data objects, the objects' manager and performs the heavy processing on the object's data. Figure 30 shows the class diagram of the business logic layer (BLL).

The BLL is composed of the following classes.

1. Article: every article object represents a part in the truck. It has an ID equal to the part number, a type and a name as a property

2. ECU: inherits the Article class and add on the following properties
   - Address: the address of the ECU used for J1939 address space communication at SCANIA[1]
   - Bus: the bus in which the ECU is located, and can be one of the following: RED, GREEN or YELLOW
   - MessageList: a list of messages that are received and transmitted
   - SW: and object representing the appl layer architecture of the ECU
   - Inhouse: a boolean property for indicating whether the ECU is in-house or not.
   - SwVer: the software version of the ECU
   - Type: the type of the ECU
   - Message: represent a message object and has the following properties
     - Priority: value of 0 has the highest priority. This value is used to control the message priority
     - Broadcast: determine if the message is transmitted as broadcast or not
     - P2P: determine if the message is transmitted as peer-to-peer (P2P).
     - DestAd: the destination address, which has a meaning only if the message is transmitted as P2P
     - PGN: Parameter Group Number refers to the value of the PDU format (PF), PDU specific (PS), the Data page bit and the reserved bit in the arbitration field of a CAN data Frame[32, p. 8]

3. SW: is an object holding appl layer architecture information for the ECU. The object is composed of the following components:
   - ANodeList: application component vertices list, where every item corresponds to a SCANIA application component in an ECU
   - RtdbNodeList: all RTDB variables in the ECU are stored in the list.
   - EdgeList: contains a list of edges linking the ACs with the RTDB variables (representing data dependencies)
In addition to the classes mentioned in the previous section, an ECU manager class take care of managing the list of all ECUs found in a truck for a given VIN number. This class implements the algorithm for abstracting the CAN bus system and generating the software architecture for the whole system. Following authors' recommendation in 5.1, the algorithms use bipartite DDG as a model for the software architecture.

To create a software architecture implementation model for each ECU, a source code parser tool is called, and its output is then parsed and used to populate a bipartite DDG. The DDG is not populated if the ECU is outsourced.

The algorithm for generating the software architecture of the whole system loops through all ECUs and checks whether it has an implementation model for its appl layer architecture or not. If a model does not exist, the ECU is assumed to be outsourced and thus a single vertex in the bipartite DDG is created. If the ECU has an implementation model, then its vertices and edges are added to the whole system bipartite DDG. When the loop ends, the obtained bipartite DDG contains all ECUs' DDG and some isolated vertices representing the outsourced ECUs.

The algorithm loops a second time through all ECUs, and this time it connects all vertices that are communicating over the CAN. It looks at every ECU's CAN message and tries to match it with a destination ECU, and if the message is broadcasted then it looks through all ECUs for a receiver. When a receiving ECU(s) is (are) found, the algorithm reads all signals in the message and connects all vertices involved in signal transmission/reception in both ECUs.

---

6 The tool is developed within the ESPRESSO project by Oscar Molin
A large part of the algorithm is dedicated to merge communicating vertices of the same type, according to the approach described in 3.1.2.2. The communicating vertices are added to a list in pairs since the algorithm connects a single sender and a single receiver at a time. Subsequently, the algorithm applies a recursive loop on the vertex-pair list to detect all communicating vertices – not only pairs. The result is a shorter list with names for new vertices that merge all communicating vertices. At the end, the algorithm removes all communicating vertices (of the same type), and replaces those with the new vertices found in the shorter list.

**Functional architecture generation**

The functional architecture generation is done in two steps. The first step is done by performing a Depth-first-search (DFS) algorithm on the DDG, and the second step by drawing the results.

1. Depth-first-search (DFS):
   Depth-first-search is performed on the DDG starting from the desired AC as the root element, then the child nodes are explored until a predefined depth is reached and then backtracking. The algorithm uses recursive implementation of the DFS, the following pseudo-code is simplification of the real implementation. Other functions used for storing the visited nodes and their relation the parent nodes are not shown here.

   ```plaintext
   function DDG(Node nSource, int depth) {
     Edge[] edges = getEdgesWhereSource(nSource)
     foreach (Edge edge in edges) {
       Node nDestination = getNodeWhereDestination(edge)
       DDG(nDestination, depth++)
     }
   }
   Node root
   DDG(root, 0)
   ```

2. The obtained structure from the first step is used to generate a JSON data structure, to be used as an input for the Data-Driven Documents (D3) JavaScript library [36] and display the graph using Collapsible Tree Layout [37]

![Figure 31: Screen-shot of functional architecture view](image-url)
**Verification of the tool**

To verify that the developed tool chain works as expected, a simpler verification has been performed by looking at its outputs.

Since an ECU's application layer may consist of many applications creating a complex communication network, it was not easy to manually verify that the tool has recreated the correct appl layer architecture for a particular ECU. A fictitious and simple ECU software architecture has instead been used as input to simplify the verification. Additionally, the algorithms that uses the information from CAN messages to connect all ECUs' implementation models and build a unified design model, has been verified using the same approach.

The generated functional design view has been verified in the same way, but also connections related to the fuel level warning functionality has been followed to check that it lines with reality.
Data access layer

This layer takes care of the persistent data, such as Spectra, Perforce repository, source code parsing servers and local databases. The data access layer is equipped with cache functionality for a quicker access to previously parsed source code. The following sections present the different modules in this layer.

Figure 32: Data Access Layer diagram, the arrows indicate the data flow direction
**Spectra module**

The Spectra module in the data access layer loads all parts for a given truck, and then extracts the existing ECUs and their software versions. This module derives data from the local database that is used to simulate access to Spectra.

<table>
<thead>
<tr>
<th>Part Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 00000018516185SSKS</td>
<td>CABLE ROUT</td>
</tr>
<tr>
<td>A 00000018516315SSKS</td>
<td>RELEASE BEARING</td>
</tr>
<tr>
<td>A 00000018516775SSKS</td>
<td>ECU SMS</td>
</tr>
<tr>
<td>A 00000018521255SSKS</td>
<td>UNION</td>
</tr>
<tr>
<td>A 00000018521265SSKS</td>
<td>PLUG</td>
</tr>
<tr>
<td>A 00000018524425SSKS</td>
<td>SCREW</td>
</tr>
<tr>
<td>A 00000018525995SSKS</td>
<td>CABLE H</td>
</tr>
<tr>
<td>A 00000018528175SSKS</td>
<td>BRAKE DISC</td>
</tr>
</tbody>
</table>

Text 3: Extract of the Input file to the Spectra Module

**ESPRESSO repository**

This part of the Data Access layer resolves data from the ESPRESSO repository, such as ECU addresses, bus type and software path (See appendix B for DTD format in the ESPRESSO repository).

**Perforce module**

The Perforce module in the Data Access layer downloads the source code of all available ECUs using the p4ftp tool. The mapping from software version to Perforce folder path is derived from the ESPRESSO repository.

**Messages loader**

The message's loader module sends the source code, for decoding and encoding CAN signals in the ECU, to the CAN parser. The CAN parser\(^7\) is a tool developed within the ESPRESSO project that builds a list of all messages and signals for every ECU. Text 4 shows a sample of the parser output which is used to populate the messages and signals objects for each ECU. See appendix B for DTD for the output format.

---

\(^7\) The CAN parser is developed by Behrooz Mokhtari and Juan Greco Arroyo

-55-
**Application component loader**

The application component loader calls a set of parsing scripts\(^8\) to extract the appl layer architecture of the current ECU. Text 5 shows an example output of the parsing scripts which represent a bipartite-graph. The graph consists of two types of vertices: AC vertices and RTDB variable vertices, in addition to, the edges between the vertices. See appendix B for DTD for the output format.

```
<node name="RTDB_HW_PART_NR_E" type="V"/>
<edge source="COO.appl.apid" destination="RTDB_HW_PART_NR_E" occurrence="1"/>
<node name="RTDB_HW_COMPLETE_NR_E" type="V"/>
<edge source="COO.appl.apid" destination="RTDB_HW_COMPLETE_NR_E" occurrence="1"/>
<node name="RTDB_HW_VERSION_NR_E" type="V"/>
<edge source="COO.appl.apid" destination="RTDB_HW_VERSION_NR_E" occurrence="1"/>
<node name="RTDB_SYSTEM_NAME_E" type="V"/>
<edge source="COO.appl.apid" destination="RTDB_SYSTEM_NAME_E" occurrence="1"/>
<node name="RTDB_ECU_SUPPLIER_NAME_E" type="V"/>
<node name="COO.appl.ccmg.ccmg" type="AC"/>
```

Text 5: Source code parser output

**Cache – intermediate storage**

The application has a caching system. Recently downloaded and parsed source code is served from the cache. The caching improves the response time of the application since many tasks required access to services located on other machines.

**3.2.2.2 Choice of the implementation technologies**

The current version of the ESPRESSO demonstrator is a desktop application. It was developed mainly in C# .NET using Microsoft Visual Studio 2012. Other languages such as HTML, JavaScript, CSS and XML were also used for user interface and persistent data storage.

Many graph visualization libraries has been tested, and also a try has been made to make own visualization library. At the end, the library D3JS [36] has been used because of the easy way to feed data to it and its interactive interface.

The ESPRESSO demonstrator has been tested on the following software and hardware configuration:

- Microsoft Windows Professional 7
- .NET Framework 4,5
- Pentium compatible PC(Intel Core i5)
- 4GB RAM

\(^8\) Created by Oscar Molin
3.3 Demonstration Results

This section summarises the results achieved in this thesis work divided into six parts.

Product model for the embedded system

A simple product model for the embedded system of a SCANIA truck has been developed; simpler than the real architecture. Since the focus in this work was on the ECUs’ appl layer architecture, the simplified embedded system model was enough for the purposes of this thesis. The model includes a description of the ECUs, path to the software, some information about the hardware and the CAN bus those are connected to. Given a particular truck, the model is populated using data from Spectra and ESPRESSO repository.

Architecture design model for the embedded software

Graph theory has been chosen to model the embedded software architecture in a SCANIA truck. Two DDG types has been developed and proposed. Three assumptions have been made so that the software architecture matches the developed model. The models adapt to the SCANIA software architecture with no need for any heavy conversions or pre-calculations. The bipartite DDG model has been tested in the ESPRESSO demonstrator tool and proved that it can be used as a model for a couple of recently produced SCANIA trucks.

AUTOSAR alignment

A straight forward mapping from the two proposed software DDGs to AUTOSAR architecture has been developed with two mapping examples presented. Mapping from SCANIA application components to AUTOSAR application layer, only highest level software layer is concerned and any CAN bus or hardware details are ignored.

In mappings according to Approach 1 – A bipartite based approach (section 3.1.2.3), the tool that reads the AUTOSAR representation is expected to be aware of implementation specific parameters in order to reconstruct the original bipartite DDG description of the truck's embedded software. The parameters tell whether an AUTOSAR SW-C is representing a shared variable or an application component. In the other case where a tool reads AUTOSAR representation according to Approach 2 – A hypergraph based approach (section 3.1.2.3), it is expected to merge input ports that are in the same SW-C and with the same hyperarc-name parameter. The result will then be the original hyper-DDG.

The mapping from ECUs' software architecture to AUTOSAR has not been implemented in the ESPRESSO demonstrator tool.

Functional architecture generation

The tool and the algorithms developed are able to trace data-dependencies between ACs to an unlimited depth9 regardless on which ECU they reside. It can be used to auto generated FAD equivalent documents but using ACs instead of the AErs. In contrast to AErs, ACs are not logical. This mean that they can be associated with an atomic software component according the ISO26262 standard [1, Sec. 1.25].

---

9 Only limited by computing resources
**ESPRESSO repository**

By implementing the ESPRESSO demonstrator tool and analysing the current SCANIA architecture, the shape of the ESPRESSO repository has been formed. For demonstration purposes, the current implementation of the repository has been partially filled with data from diverse documents at SCANIA. The current version of ESPRESSO repository contains the following info:

- All used ECU types at SCANIA with their:
  - Name
  - CAN addresses
  - Connected main CAN bus
- Mapping from UF to a SCANIA application component
- Source code paths in Perforce for a specific ECU type and software version

**ESPRESSO demonstrator tool**

The developed ESPRESSO demonstrator tool was the implementation of the concept presented in 3.2.2. It showed how both the embedded system and the embedded software models could be applied on two sample trucks and used to generate functional architecture view. It also proved that the recovery task could be automated if data sources at SCANIA strictly hold a specific format and access is guaranteed to all of them. The tool demonstrates an example implementation of the ESPRESSO repository and what data that needs to be stored there.

The ESPRESSO demonstrator has been shown in a SCANIA internal fair and in divers meetings where it has been valued and got much attention; it was a success for the ESPRESSO project, according to the project manager Mattias Nyberg.
This Chapter discusses the achieved results and the taken approaches as well as suggesting future work.

4.1 Discussion

The designed concept was general and modular, which makes it easy to build upon it and reuse it for other purposes by introducing small changes. Despite its generality, the concept fulfilled SCANIA needs. The following sections discuss various aspects and choices in the applied solution in more details. Specifically, the two first sections discuss the two questions presented in the Purpose section.

Functional architecture recovery

The hardness of automating the access to data sources at SCANIA caused a large overhead for the progress of the thesis work. Many tasks were done manually, which took much longer time than expected. Structuring the current/future stored data in SCANIA would contribute to a more efficient work flow when fetching it, and will open new possibilities to automate tasks that are manually performed today.

To be able to auto generate the functional architecture of a SCANIA truck's embedded software, information stored in databases must have a well-defined structure, which could be parsed. To enforce the database compliance to a given structure, entries in the databases should be taken from a pre-defined set of values.

In addition, source code should also follow a structure that simplifies its parsing and matches the proposed architecture implementation model of the embedded software. As an example, the code should be separable into non-overlapping application components which should only communicate through shared variables. However, enforcing and checking the compliance with coding style rules could be difficult and complicated and needs cooperation from different development teams.

The ESPRESSO repository

The ESPRESSO repository is meant to replace existing data sources that are in a non-machine readable format or data that is not explicitly expressed formally anywhere. It means that data should be entered directly into the repository to bypass manually created documents.

The proposed content of the repository should be seen as a first version, and there will be a future need to add more contents such as functional requirements related information.

The relevance of the assumptions

The following paragraphs discuss whether the assumptions taken in this thesis were relevant and applicable to a SCANIA truck's embedded software or not.

Assumption 1 (section 3.1.2.1): The current SCANIA software implementation does follow this assumption in the most cases, but there are some exceptions. According to the system developers at SCANIA, there are always reasons to take direct communication between application components especially in critical systems where extra overhead is unwanted.

Assumption 2 (section 3.1.2.2): This assumption is not radical, and maybe already applicable to the existing code at SCANIA. That is because almost always, in a single ECU, the shared variables
containing the received data are different from those containing data to be transmitted, as well as that the data received from one CAN message is stored in a dedicated shared variable (two received messages are not sharing the same variable); thus breaking any possible chain communication and preventing any possible problematic connection.

**Assumption 3** (section 3.1.2.2): Writing to the same variable both by an internal AC and by an external writer is in many cases useless or undesired; often there is a need to keep received data untouched. So again, this assumption is neither radical nor hard to fulfil, and most probably already applies in the current SCANIA architecture.

**CAN sub-buses**

The product model for embedded system lacks the support for logical CAN sub-buses between a host physical ECU and its guest logical ECU. To support those sub-buses the model needs to be extended to distinguish between logical and physical ECUs and to assign gateway functionality to the host ECU.

**Representation of the real product**

The proposed architecture models are a simplification of the reality, and their correctness depends on the product specification data. In this work, it has been tried to collect data from data sources that defines the product in the best possible way. However, it was not easy to get access to all ECU source codes (read more in section 3.2.1.2). In addition, End-Of-Line (EOL) parameters\(^\text{10}\) were not taken into consideration. Despite these obstacles, a close description of the product could be reached using the proposed model, which was verified by among others comparing to the already known architecture for the Fuel level warning UF.

**Limitation in the proposed architecture design model**

The proposed approach to unify all ECU software DDGs uses information from CAN messages to merge shared variables. However, there is nothing that guarantees that merged shared variables are updated immediately when the content of at least one of those is modified. There is no rule saying that a CAN message should be transmitted as soon as a shared variable has changed its content (to update reading shared variables). Looking at figure 20, if AC3 writes new content to V2, and ECU1 does not transmit a CAN message to update V3, then AC2 and AC4 will not read the same content from V2 and V3 respectively.

This discussion implies that the model is suitable to model data dependencies but not data-flow as a function of time.

**The Embedded software model usage in other areas**

The developed embedded software architecture design mode is general so that it can be used in other areas where the software can be divided into separable non-overlapping entities. The software entities shall exclusively communicate through shared resources (e.g. storage, services etcetera). The modelled software does not need to be embedded to fit to the model.

**AUTOSAR standardisation**

By mapping the developed software model to AUTOSAR, the model description is standardised and can be easily exchanged with other communities for purposes such as comparisons, evaluations or compliance checks.

\(^{10}\) End-of-line parameters are parameters that are set at the end of the production. The ECU software behaves differently dependent on the values of these parameters.
**ESPRESSO demonstrator software architecture design**

The chosen software architecture for the ESPRESSO demonstrator tool makes it possible to apply changes to a specific layer without affecting the whole program. At the current stage, the program was intended to be a prototype demonstrator as a proof of concept. With the actual software architecture, it is easy to take the prototype as it is and only develop the relevant layers so that it meets any future requirements.

**Verification of the models and the tool**

During this thesis work, only basic manual verification of the results has been performed. The verification was mainly done on the produced outputs from the tool. However, a thoroughly verification of the whole tool chain including all algorithms as well as the resulting recovered architecture should be done before using it in production.

The architecture of the application layer in a single ECU is so simple so that the proposed models do not need simulations or tests to verify if it fits to the implemented software architecture. On the other hand, more effort should be put into investigating possible scenarios that may cause incompatibility or wrong recovery of the software architecture when abstracting away the ECU allocation information (unifying all ECUs' DDGs) and unifying shared variables that are located in different ECUs. In this thesis, three issues have been retrieved, but without further investigations there is no guarantee that no more issues exist.

Different parts of the tool chain has been partially verified manually, but to ensure that it follows the proposed models and methods, the tool output should be verified carefully and compared to the implemented software architecture. Nonetheless, any verification results and detected issues should not impugn the principle of functional architecture auto-creation. Therefore, verification was not considered to have the main importance in this thesis.
4.2 Future work

Since the embedded system architecture implementation/design model is expressed as a DDG, graph based analyses can be applied on it. As examples of analysis that are relevant to the embedded system, the following can be stated: data-flow penalties, critical path calculations, allocation and time optimization and CAN-bus load analysis. Scheduling and allocation information can also be used to estimate the power consumed by the ECUs in different running modes of the embedded system.

An area which can be of a very interest for SCANIA is to evaluate different scheduling algorithms/orders of the application components by easily using the DDG. An alternative would be to simply calculate the optimal schedule by applying algorithms such as extended retiming [38].

Allocation of the application components to different ECUs can be a hard task to perform manually. However, auto-optimising with respect to power, timing, scheduling and allocation can be done using allocation graphs [39] by evaluating all possible paths in the tree.

When it comes to safety, the model can be used to check different allocation alternatives’ with regards to the ASIL decomposition [40] and whether they follow the rules and requirements according to the ISO26262 standard.

The work done in this thesis did not include sensors and actuators connected to the ECU. To get a complete picture of the data dependencies, it is essential to trace data flowing from/to peripherals. This would require modifications in the architecture model of the embedded system, the algorithm generating the DDGs and the AUTOSAR mapping. The ESSPRESSO repository would also need expansion to accommodate the changes.

As discussed in the previous section, to use the models/tool in production more effort should be put in verification. Also, EOL parameters should be available and used to recover correct software architecture. The EOL parameters should affect both the source code parser and the CAN parser which provide input to the ESPRESSO demonstrator.
5 RECOMMENDATIONS AND CONCLUSIONS

This Chapter states some recommendations based on the results of this thesis work and draws some conclusions.

5.1 Recommendations

The authors of this thesis recommend the usage of bipartite DDG as a model for the appl layer architecture in SCANIA's ECUs. The reason to prefer the bipartite DDG over hyper-DDG is the availability of more algorithms designed for bipartite graphs, and that it is easier to map to AUTOSAR representation of a vehicle’s embedded software (see 3.1.2 for details). Additionally, since the shared variables are data holders and deliverers, the bipartite DDG interpretation of those as (communication) vertices is more natural than interpreting those as hyperarcs (in a hyper-DDG).

The authors also recommend that SCANIA starts a process to make their data more consistent and easier to access. The effort put on this process will be gained back in form of higher productivity and more automated tasks.

5.2 Conclusions

A truck at SCANIA is defined by data specifications stored in multiple databases, source code files and manually created documents. Without introducing any changes, automating the recovery of the functional architecture from the existing infrastructure and data sources at SCANIA is difficult because of the following:

- Information missing or stored in a non-machine readable format.
- The lack of well-defined naming conventions for database entries
- Changes in folder structures of the source code depending on the ECU and the software version.

The work done in this thesis shows that by introducing minor changes to the current data structure, programming style and architecture, it is possible to automate the recovery of the functional architecture. Information that is not explicitly documented or is documented in a machine non-readable format has been recovered and proposed as content for the ESPRESSO repository.

The proposed architecture design model for the embedded software has been used to recover the functional architecture. The same model can also be used to auto generate other documents and work products related to the development process, analysis, reports etcetera.
6 REFERENCES

DEPENDENCY GRAPHS,” University of Texas, Austin, Texas, 1988.


[33] Rosam IT-Tjänster AB, Rosam. Sweden: Rosam IT-Tjänster AB.


APPENDIX A: Calculating the number of AUTOSAR ports – formula proof

Given a directed hypergraph $G$ with an $n \times m$ incidence matrix $H$, then it is known that the graph consists of $n$ hyperarcs and $m$ vertices.

A row $i$ in the incidence matrix represents a single hyperarc where the value of each element $h_{ij}$ can either be:

- $0 \equiv$ vertex $j$ is non-connected
- $1 \equiv$ vertex $j$ is a target vertex
- $-1 \equiv$ vertex $j$ is a source vertex

Then, the number of source vertices to the current hyperarc can be calculated by subtracting the difference in the number of source and target vertices in row $i$ from the total number of all vertices in the same row. The result of the subtraction will be twice the number of source vertices, and therefore should be divided by 2. The difference in the number of source and target vertices can be easily calculated by summing all elements in the row. The total number of all vertices in the current row can be found by summing the absolute value of all elements in the row; see formula 4.

$$\sum_{j=1}^{m} |h_{ij}| - \sum_{j=1}^{m} h_{ij} = \sum_{j=1}^{m} |h_{ij}| - h_{ij} \quad (4)$$

A similar approach is taken to calculate the number of target vertices, the only difference is that instead of calculating the difference, the sum is calculated; see formula 5.

$$\sum_{j=1}^{m} |h_{ij}| - \sum_{j=1}^{m} h_{ij} = \sum_{j=1}^{m} |h_{ij}| \quad (5)$$

When splitting a hyperarc in approach 2, new edges are created with a single source vertex and an arbitrary number of target vertices, thus only increasing the number of target ports by the number of source ports. If a hyperarc $i$ have $s$ source vertices and $t$ target vertices, then the total number of ports connected to all split edges can be calculated according to formula 6.

$$s + (s \times t) = s \times (1 + t) \quad (6)$$

The total number of AUTOSAR ports after mapping the complete hypergraph according to approach 2 can be calculated by inserting formula 4 and 5 into formula 6 and summing over all rows (representing hyperarcs); see formula 7.

$$\sum_{i=1}^{n} \left[ \sum_{j=1}^{m} \frac{|h_{ij}| - h_{ij}}{2} \right] \times \left( \sum_{j=1}^{m} \frac{|h_{ij}| + h_{ij} + 1}{2} \right) = \sum_{i=1}^{n} \left[ \sum_{j=1}^{m} \frac{|h_{ij}| - h_{ij}}{2} \right] \times \left( \sum_{j=1}^{m} \frac{|h_{ij}| + h_{ij} + 1}{2} \right) \quad (7)$$
APPENDIX B: Document type definition DTD

This appendix contains DTD for different XML files, where some of them are part of the ESPRESSO repository.

Mapping from UF to AC

```xml
<!DOCTYPE root[ <!-- root element -->
  <!ELEMENT root (UF+)> <!-- UF element -->
  <!ELEMENT UF (#PCDATA)> <!-- UF full name -->
  <!ATTLIST UF id ID #REQUIRED> <!-- Unique id -->
  <!ATTLIST UF code CDATA #REQUIRED> <!-- UF code -->
  <!ATTLIST UF ca CDATA #REQUIRED> <!-- UF category -->
  <!ATTLIST UF ac CDATA #REQUIRED> <!-- Mapped AC -->
]>
```

Message list

```xml
<!DOCTYPE root [ <!-- root element -->
  <!ELEMENT root (message+)> <!-- message -->
  <!ELEMENT message (name, PGN, SourceAd, DestinationAd, Priority, Proprietary, P2P, Broadcast, signals)> <!-- message -->
  <!ELEMENT name (#PCDATA)> <!-- message name -->
  <!ELEMENT PGN (#PCDATA)> <!-- message PGN -->
  <!ELEMENT SourceAd (#PCDATA)> <!-- message SourceAd -->
  <!ELEMENT DestinationAd (#PCDATA)> <!-- message DestinationAd -->
  <!ELEMENT Priority (#PCDATA)> <!-- message Priority -->
  <!ELEMENT Proprietary (#PCDATA)> <!-- message Proprietary -->
  <!ELEMENT P2P (#PCDATA)> <!-- message P2P -->
  <!ELEMENT Broadcast (#PCDATA)> <!-- message Broadcast -->
  <!ELEMENT signals (signal+)> <!-- signal elem -->
  <!ELEMENT signal (name, rtdbid, startbit, bitlength, unit)> <!-- signal -->
  <!ELEMENT name (#PCDATA)> <!-- signal name -->
  <!ELEMENT rtdbid (#PCDATA)> <!-- signal rtdbid -->
  <!ELEMENT startbit (#PCDATA)> <!-- signal startbit -->
  <!ELEMENT bitlength (#PCDATA)> <!-- signal bitlength -->
]>
```

Embedded System Model

```xml
<!DOCTYPE root [ <!-- root element -->
  <!ELEMENT root (ecus?)> <!-- ecus -->
  <!ELEMENT ecus (ecu*)> <!-- ecus -->
  <!ELEMENT ecu (message+)> <!-- ecus -->
  <!ELEMENT ecu (SW*)> <!-- ecus -->
  <!ELEMENT message (name, PGN, SourceAd, DestinationAd, Priority, Proprietary, P2P, Broadcast, signals)> <!-- ecus -->
  <!ELEMENT name (#PCDATA)> <!-- ecus -->
  <!ELEMENT PGN (#PCDATA)> <!-- ecus -->
  <!ELEMENT SourceAd (#PCDATA)> <!-- ecus -->
  <!ELEMENT DestinationAd (#PCDATA)> <!-- ecus -->
  <!ELEMENT Priority (#PCDATA)> <!-- ecus -->
  <!ELEMENT Proprietary (#PCDATA)> <!-- ecus -->
  <!ELEMENT P2P (#PCDATA)> <!-- ecus -->
  <!ELEMENT Broadcast (#PCDATA)> <!-- ecus -->
]>
```
Software Model

ECU Software Data-base

-70-
APPENDIX C: An example AUTOSAR SW-C

<!--SW-C architecture-->  
<APPLICATION-SOFTWARE-COMPONENT-TYPE>
  <SHORT-NAME>SampleComponent</SHORT-NAME>
  <PORTS>
    <R-PORT-PROTOTYPE>
      <SHORT-NAME>SampleRPORT</SHORT-NAME>
      <REQUIRED-COM-SPECS>
        <UNQUEUED-RECEIVER-COM-SPEC>
          <DATA-ELEMENT-REF DEST="DATA-ELEMENT-PROTOTYPE">
            PathToDataElement_1
          </DATA-ELEMENT-REF>
        </UNQUEUED-RECEIVER-COM-SPEC>
      </REQUIRED-COM-SPECS>
      <REQUIRED-INTERFACE-TREF DEST="SENDER-RECEIVER-INTERFACE">
        PathToRemoteReceiverPort
      </REQUIRED-INTERFACE-TREF>
    </R-PORT-PROTOTYPE>
  </PORTS>
</APPLICATION-SOFTWARE-COMPONENT-TYPE>

<!--SW-C implementation description-->  
<SWC-IMPLEMENTATION>
  <SHORT-NAME>SampleComponentImpl</SHORT-NAME>
  <BEHAVIOR-REF DEST="INTERNAL-BEHAVIOR">
    PathToInternalBehaviourAUTOSARTagDescribingInternalBehaviour
  </BEHAVIOR-REF>
</SWC-IMPLEMENTATION>

<!--SW-C internal behaviour-->  
<INTERNAL-BEHAVIOR>
  <SHORT-NAME>SampleComponentInternalBhv</SHORT-NAME>
  <COMPONENT-REF DEST="APPLICATION-SOFTWARE-COMPONENT-TYPE">
    PathToTheSW-CToWhichThisBehaviourBelong
  </COMPONENT-REF>
  <EVENTS>
    <TIMING-EVENT>
      <SHORT-NAME>SampleComponentMainTrigger</SHORT-NAME>
      <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">
        Runnable_1
      </START-ON-EVENT-REF>
      <PERIOD>1.0</PERIOD>
    </TIMING-EVENT>
  </EVENTS>
  <RUNNABLES>
    <RUNNABLE-ENTITY>
      <SHORT-NAME>Runnable_1</SHORT-NAME>
      <CAN-BE-INVOKED-CONCURRENTLY>false</CAN-BE-INVOKED-CONCURRENTLY>
      <DATA-READ-ACCESS>
        <DATA-READ-ACCESS>
          <SHORT-NAME>ReadingFromSampleRPORT</SHORT-NAME>
          <R-PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">
            PathToSampleRPORT
          </R-PORT-PROTOTYPE-REF>
          <DATA-ELEMENT-PROTOTYPE-REF DEST="DATA-ELEMENT-PROTOTYPE">
            PathToDataElement_1
          </DATA-ELEMENT-PROTOTYPE-REF>
        </DATA-READ-ACCESS>
      </CAN-BE-INVOKED-CONCURRENTLY>
      <RUNNABLES>
        <SUPPORTS-MULTIPLE-INSTITANTIATION>false</SUPPORTS-MULTIPLE-INSTITANTIATION>
      </RUNNABLES>
    </RUNNABLE-ENTITY>
  </RUNNABLES>
</INTERNAL-BEHAVIOR>