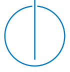


# Model-Based Development of Software-intensive Automotive Systems

**Stefan M. Kugele**

TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Software & Systems Engineering

# Model-Based Development of Software-intensive Automotive Systems

**Stefan M. Kugele**

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

**Doktors der Naturwissenschaften (Dr. rer. nat.)**

genehmigten Dissertation.

**Vorsitzender:** Univ.-Prof. Dr. Hans-Joachim Bungartz

**Prüfer der Dissertation:**

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Helmut Veith, Technische Universität Wien / Österreich

Die Dissertation wurde am 16.07.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 06.11.2012 angenommen.

# Abstract

The automotive industry in general and the automobile in particular changed fundamentally during the last 125 years. First automobiles reflected the pioneering spirit of their constructors in the field of classical engineering disciplines. During the last 40 years, however, a shift from a few electrical control units with dedicated functions to complex networks of highly interconnected, distributed, and multi-functional software-intensive systems took place. This change was accomplished at a pace engineers, engineering tools, and processes could not follow. As a consequence, the expected and also targeted high quality and reliability standards could not be reached in any case. New methodologies and tools have to be developed in order to cope with future challenges—electromobility, environmental compatibility, and sustainability are only few of them.

This thesis proposes the 'COLA automotive approach'. It enables seamless model-based development of software-intensive automotive systems along different levels of abstraction. This leads to a reduction of complexity and takes the principle of separation of concerns into account. The both syntactical and semantical well-defined domain-specific modelling language COLA is used to design such systems and hence is the basis for further activities. For example, model analysis, model transformation, and semantics preserving deployment can be performed. The capabilities of the COLA-IDE are completed by an export of simulatable requirements specification documents and a traceability analysis. The COLA engineering tool stores all artefacts in a single repository ('single point of truth') reflecting the product data model, hence avoiding tool integration gaps. By means of several case studies, the approach has been evaluated and its feasibility has been shown.

# Kurzzusammenfassung

Die Automobilindustrie im Allgemeinen und das Automobil im Besonderen unterlagen während der vergangenen 125 Jahre weitreichenden Veränderungen. Die ersten Fahrzeuge spiegelten den Pioniergeist ihrer Konstrukteure speziell in klassischen Ingenieursdisziplinen wider. Während der letzten 40 Jahren fand jedoch eine Weiterentwicklung statt von einigen wenigen Steuergeräten mit sehr spezifischen Funktionen hin zu sehr komplexen untereinander stark vernetzen und verteilten softwarelastiger Systeme. Dieser Wechsel vollzog sich mit einer derartigen Geschwindigkeit, dass Ingenieure, Werkzeuge und Prozesse dieser Entwicklung nicht schritthalten konnten. Als Konsequenz konnten die erwarteten und angestrebten hohen Qualitäts- und Zuverlässigkeitsstandards nicht immer erreicht werden. Neue Vorgehensweisen und Werkzeuge müssen entwickelt werden um den zukünftigen Herausforderungen wie Elektromobilität, Umweltverträglichkeit und Nachhaltigkeit gerecht zu werden.

Diese Arbeit schlägt den „COLA Automotive Ansatz“ vor. Dieser ermöglicht eine durchgängige modellbasierte Entwicklung von softwareintensiven Automotivesystemen entlang unterschiedlicher Abstraktionsebenen. Diese führen zu einer Reduzierung der Komplexität und tragen dem Prinzip „Separation of Concerns“ Rechnung. Die sowohl syntaktisch als auch semantisch formal definierte domänenspezifische Modellierungssprache COLA dient zur Beschreibung dieser Systeme und bildet somit das Fundament für weitergehende Aktivitäten. So können beispielsweise Modellanalysen, Modelltransformationen und ein semantikerhaltendes automatisches Deployment durchgeführt werden. Eine Ausleitung simulierbarer Lastenhefte sowie eine Nachverfolgbarkeitsanalyse runden die Fähigkeiten der COLA-IDE ab. Das COLA Entwicklungswerkzeug speichert alle Artefakte in einer zentralen Datenbank („Single Point of Truth“), welche das Produktdatenmodell abbildet und somit Lücken bei der Werkzeugintegration vermeidet. Anhand mehrerer Fallstudien wurde der Ansatz evaluiert und dessen Tragfähigkeit bestätigt.

*Geniale Menschen beginnen große Werke, fleißige Menschen vollenden sie.*

—Leonardo da Vinci (1452-1519)

# Acknowledgements

I would like to thank all the people who helped me to make this dissertation a success. First, I want to express my gratitude to my supervisors Manfred Broy and Helmut Veith for giving me the opportunity to work at their chairs. Moreover, I want to thank Javier Esparza and his group for providing me such an inspiring and cordial working environment.

In addition, my thanks also go to all the people with which I had the pleasure to work over the last years in different projects. In particular to my workmates Wolfgang Haberl, Markus Herrmannsdörfer, Stefano Merenda, Sabine Rittmann, Bernd Spanfelner, Michael Tautschnig, Zhonglei Wang, and Doris Wild who worked hand in hand in the BASE.XT and BASE.XT Pro Live projects and made it to such a success story. At this point I would particularly like to thank Martin Wechs from BMW Group, who supported the project over years and touted for both the project and the gained research results within the company.

This dissertation is based on a number of already published papers. I am grateful to the co-authors which supported me to publish these papers. Not abovementioned are Andreas Holzer, Visar Januzaj, and Christian Schallhart whom I am indebted to, as well. I would also like to thank the reviewers that tremendously helped me to improve the dissertation with their comments.

I am deeply grateful to Farhad, my family—especially my mother—, and friends who supported me over the years. They gave me the strength and motivation to achieve my aims and to overcome setbacks.

Munich, July 2012 *Stefan Kugele*

# CONTENTS

**1. Introduction** **1**
1.1. Initial Situation . . . 1
1.2. Approach . . . 5
1.3. Contributions of this Thesis . . . 6
1.4. Outline of this Thesis . . . 9

**2. Background** **13**
2.1. Embedded Systems: An Overview . . . 13
2.2. Historical Review . . . 17
2.2.1. Classical Mechanical Engineering . . . 17
2.2.2. From Revolution to Disenchantment of Software . . . 18
2.3. Automotive Industry Characteristics . . . 23
2.3.1. Automotive Industry . . . 24
2.3.2. Automotive Domains . . . 28
2.4. Current and Future Challenges . . . 33
2.4.1. Heterogeneity . . . 33
2.4.2. Clash of Cultures . . . 35
2.4.3. Control the Complexity . . . 36
2.4.4. Move from E/E Component-driven Development Towards Function- and Mode-driven Development . . . 38

**3. Seamless Model-Driven Automotive System Development** **41**
3.1. Introduction . . . 42
3.2. Separation of Concerns Through Abstraction and Modularisation . 42
3.3. Theoretic Foundation: A Fertile Soil for Formal Methods . . . 46
3.4. From Isolated Tools to an Integrated Authoring Environment . . . 48
3.4.1. Daily Practice . . . 48

3.4.2. Solution . . . 51
3.5. Summary . . . 53

**4. The COLA Automotive Approach** **55**
4.1. Introduction . . . 55
4.2. Architectural Levels . . . 56
4.2.1. Feature Architecture . . . 56
4.2.2. Logical Architecture . . . 60
4.2.3. Technical Architecture . . . 61
4.2.4. Summary . . . 65
4.3. COLA—The Component Language . . . 66
4.3.1. Basic Concepts . . . 67
4.3.2. Operating Modes . . . 71
4.3.3. Syntax and Semantics of COLA . . . 74
4.3.4. Examples from Control Theory . . . 75
4.4. Deployment Process . . . 77
4.5. Related Work . . . 80
4.5.1. Modelling Along Different Levels of Abstraction . . . 80
4.5.2. Behavioural Modelling . . . 82

**5. Model Analysis** **85**
5.1. Introduction . . . 85
5.2. Requirements Analysis . . . 86
5.2.1. Introduction . . . 87
5.2.2. Realisation . . . 89
5.2.3. Discussion . . . 91
5.3. Deterministic Models . . . 92
5.3.1. Introduction . . . 92
5.3.2. Problem . . . 93
5.3.3. Realisation . . . 97
5.4. COLA Model Analysis via a Translation to Coloured Petri Nets . . . 101
5.4.1. Introduction to Coloured Petri Nets . . . 101
5.4.2. Translation Schema . . . 104
5.4.3. Translation Algorithm . . . 111
5.4.4. Example . . . 111
5.4.5. Related Work . . . 114
5.4.6. Summary . . . 115

**6. Generation of Requirements Specification Documents** **119**
6.1. Introduction . . . 120
6.2. Document Structure . . . 121
6.2.1. Differentiation between Customer- and System Requirements 122
6.2.2. Structuring . . . 123
6.3. Realisation . . . 126
6.4. Semantically Tangible Requirements Documents . . . 129
6.5. Integration into the Context of IEEE Std 830-1998 . . . 131
6.6. Summary . . . 132

**7. Deployment** **135**
7.1. Introduction . . . 135
7.2. Allocation . . . 139
7.2.1. Notation . . . 139
7.2.2. Constraints . . . 141
7.2.3. Realisation . . . 146
7.3. Scheduling . . . 147
7.3.1. Terminology . . . 147
7.3.2. A Taxonomy of Real-Time Scheduling Algorithms . . . 148
7.3.3. Dependency Analysis . . . 150
7.3.4. Constraint System . . . 160
7.3.5. Realisation . . . 163
7.3.6. Complexity Analysis . . . 166
7.4. Fault Tolerance . . . 169
7.4.1. Fault Hypothesis . . . 170
7.4.2. Adaptions . . . 172
7.5. Related Work . . . 174
7.6. Summary . . . 179

**8. Case Studies** **181**
8.1. Adaptive Cruise Control . . . 182
8.1.1. Functional Description . . . 183
8.1.2. Hardware Topology and Execution Platform . . . 184
8.1.3. Summary . . . 185
8.2. Autonomous Parking System . . . 185
8.2.1. Functional Description . . . 187
8.2.2. Hardware Topology . . . 188
8.2.3. Execution Platform . . . 190

8.2.4. Summary . . . 191
8.3. Comfort Hatchback Opener . . . 191
8.3.1. Functional Description . . . 192
8.3.2. Requirements Specification Documents . . . 194
8.3.3. Chain of Effects . . . 194
8.3.4. Summary . . . 195
8.4. Summary . . . 195

**9. Summary and Outlook** **197**
9.1. Summary . . . 197
9.2. Outlook . . . 199

**Bibliography** **203**

**List of Figures** **225**

**List of Algorithms** **227**

**List of Tables** **229**

**Glossary** **231**

**A. PID Controller** **237**
A.1. COLA model . . . 237
A.2. Simulation with different parameter sets . . . 237

CHAPTER

# ONE

# Introduction

The topic of this thesis is model-based development of software-intensive automotive systems. The work at hand reports on the COLA automotive approach, which considerably contributes to the state-of-the-art and state of today's practice. We first explain the initial situation by outlining the current state of practice in automotive software development in Section 1.1. Next, in Section 1.2 the basic ideas of the COLA automotive approach are sketched. In Section 1.3, we list the major contributions of this dissertation and finally give an outline in Section 1.4.

**Contents**


1.1. Initial Situation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2. Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.3. Contributions of this Thesis . . . . . . . . . . . . . . . . . . . . . . 6
1.4. Outline of this Thesis . . . . . . . . . . . . . . . . . . . . . . . . 9


## 1.1. Initial Situation

**Technological View.** Development of software-intensive automotive systems changed fundamentally within a relatively short period of time. After the first functions were realised using software about 40 years ago, a multitude of thitherto unimaginable features entered the automobile mainstream in the proceeding decades. This change was accomplished at a pace engineers, engineering tools, and processes could not follow. As a consequence, the expected and also targeted high quality and reliability standards could not be reached in any case. This is

reflected in the warranty costs carmakers have to pay each year. To take a single example: in 2000, DaimlerChrysler paid USD 1.5 billion warranty costs because of quality issues of the Mercedes-Benz luxury-car. These costs amount to a total of up to USD 500 per vehicle [14]. According to IBM research, about 30% of these costs are attributable to software and electronics defects. Against this background and an evermore penetration of E/E (**E**lectrical/**E**lectronic) systems in vehicles, quality assurance along the value-added chain is of vital concernment.

The constantly increasing complexity of automobiles itself but also the complexity to develop them can be attested to be the crucial point of automotive software engineering. The mentioned pace of change was accomplished in different di-

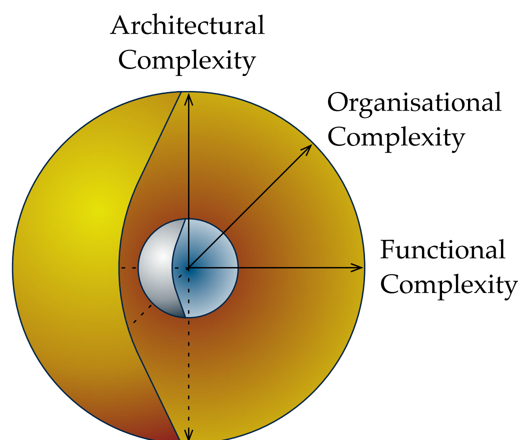



**Figure 1.1.:** Multi-dimensional explosion of complexity.

rections (visualised with three criteria in Figure 1.1). Nowadays, more than 3000 software controlled functions (→ *Functional Complexity*) perform a multitude of control and monitoring tasks. Of course, only some of them are actually visible to customer. This tremendous increase in functionality is reflected in the growth of the E/E architecture both in the number of **E**lectronic **C**ontrol **U**nits (ECUs) and the number of technological different bus systems (→ *Architectural Complexity*). In 1968, Conway [49] formulated a famous law named after him postulating that the structure of a product reflects that of the company's organisation. This has decisive impacts on the software with respect to the overall engineering effort in general and maintainability in particular. Many automotive companies are still organised in terms of responsibilities for ECUs and not in terms of functionality. This cannot be future-proof with regard to integration of functions, i. e., many functions will be executed on less ECUs. Hence, parts of different functions are mixed on a single ECU, which makes coordination work a difficult job (→ *Organisational Complexity*).

**electronic control unit**

Up to now, there is to the best of our knowledge no commercial tool which seamlessly captures the whole product life-cycle process in an adequate way. Rather, the tool landscape is best characterised as an ad hoc coupled tool chain with in many cases incompatible product data models. Different product data models and oftentimes also different semantics when considering for example tools like MATLAB/Simulink/Stateflow, ASCET, and SCADE hamper the use of formal methods from a complete vehicle perspective.

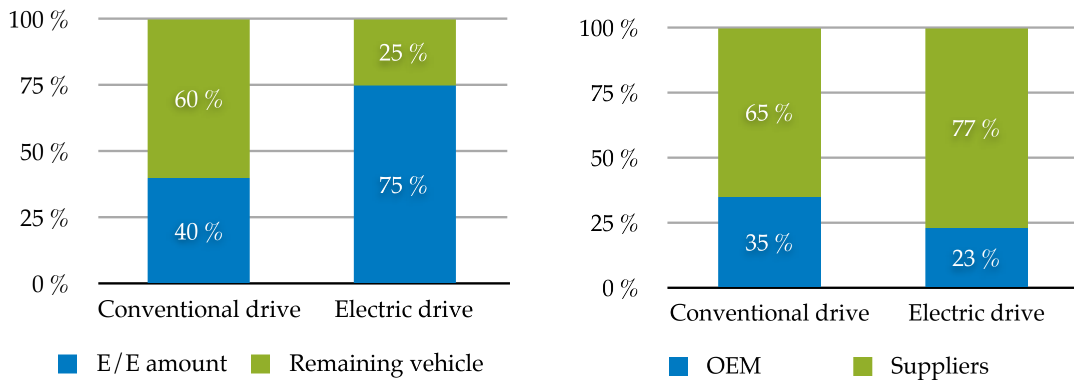
Model-based development or model-based engineering is a promising development methodology and in many disciplines a de-facto standard. The pervasive use of models enables a sufficient raise of abstraction facilitating to grasp even huge models. Furthermore, models with a sufficient amount of information and details can be used to synthesise code for the target hardware for instance.

**Macroeconomic View.** The German industry is characterised by the automotive sector unlike any other country of the European Union and even worldwide [137]. According to Bernard et al. [23], the automotive industry contributed in 2009 with about 20% (> 263 billion euros) to the German total sales. With about 723.000 employees, the automotive industry is one of the most important employers in Germany. When also considering the dense network of the supplying industry, more than five million jobs are either directly or indirectly dependent on automobiles. Denner [55] estimates that the added value for E/E components in electric vehicles will increase from 40% to 75% in contrast to cars with conventional drive. This increase in turn will reduce the added value of OEMs (**O**riginal **E**quipment **M**anufacturer) from 35% to 23% (cf. also Figures 1.2a and 1.2b). Eberbach-Sahillioglu [66] estimates that the worldwide added value of automotive electrics and electronics will increase up to 316 billion euros till 2015.

**original equipment manufacturer**

Today, the German semiconductor market for motor vehicle electronics is more than 4.1 billion euros. Automobile electronics has a share of more than 41% of the total German market for electronic components with a growth of about 5% in 2012 [39]. Burkert concludes from these numbers the importance of electric and electronic systems in vehicles. In order to maintain technological leadership, it is important to accomplish efforts to reduce E/E architectural complexity (for example by a high level of integration in electronic components) with rigorous systems engineering in general and automotive software engineering in particular. Staying the leading driver for automotive innovations is a precondition to preserve and create stainable jobs.

The importance of automotive electronics in general and software in particular is shown in a study done by Mercer Management Consulting and HypoVereinsbank




**(a)** Added value w.r.t the drive forms.

**(b)** Distribution of the added value between OEM and suppliers.

**Figure 1.2.:** Figure (a) shows the shift of the added value with respect to the E/E parts between conventional and electric drive. Figure (b) illustrates that the added value amount of suppliers will even more increase in case of electric drives (according to Denner [55]).

in 2001 [149]. Accordingly, in the year 2010, 13% of a car's production cost is associated to software. Together with the electronic components the software runs on, 35% are reached. This is similar to Grimm [83] who expects about 40% of the overall production cost for electronics (cf. Figure 1.3). Interestingly, the proportion

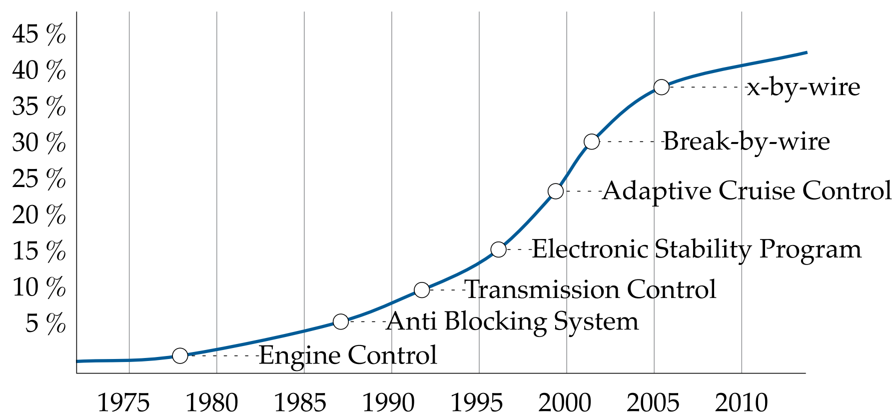



**Figure 1.3.:** Proportion of cost incurred by electronics [83].

of costs dedicated to software and hardware changed from 20% : 80% in the year 2000 to 38% : 62% in 2010, respectively (cf. Figure 1.4). Again, this underlines the

importance of software. According to Grimm, Daimler expects that about 80% of

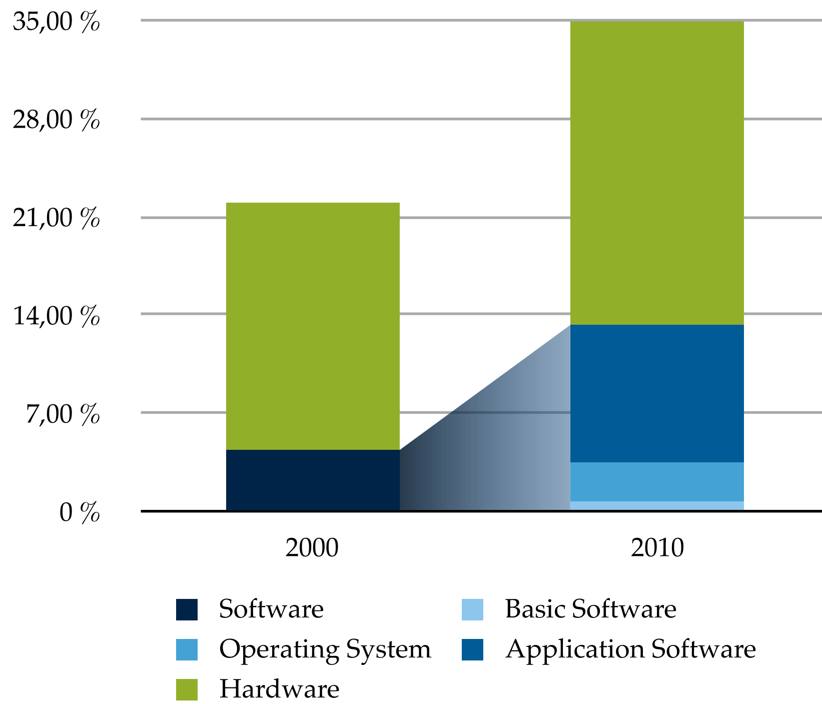



**Figure 1.4.:** E/E value of software and hardware.

all future innovations will be electronics-driven and 90% thereof by software.

This thesis makes a significant contribution in the field of automotive software engineering by proposing the COLA *automotive approach* sketched next.

## 1.2. Approach

In full awareness of the complexity, the COLA automotive approach contributes to the state-of-the-art and state of practice in the field of model-based development of software-intensive automotive systems. The centrepiece of the COLA automotive approach is the well-defined graphical and textual modelling language COLA—The Component language, which enables the application of formal methods. COLA models are designed along different levels of abstraction, namely the *Feature Architecture*, the *Logical Architecture*, and the *Technical Architecture*. Artefacts modelled on the former are the most abstract ones, whereas the Technical Architecture shows the most concrete, i. e., technical artefacts close to the execu-

tion platform. All artefacts and their relations are stored in a central repository reflecting the product data model. This allows to run verification or analysis tasks 24 hours a day, 7 days a week, and decouples these or similar time-consuming tasks from the engineer's workstation. One is aspired to perform as many analysis tasks at an early stage of the development process ('front loading'). The COLA automotive approach follows the idea of a *systems compiler*, which means that an executable system is generated directly from the model under consideration of optimisation goals and constraints. The COLA-IDE as a sophisticated authoring and engineering tool is used to model automotive E/E systems both graphically and textually. It has the distinction of being a fully integrated engineering tool rather than being characterised as an ad-hoc coupled pragmatic tool chain. This seamless integration of modelling, analysis, and deployment capabilities renders the deployment procedure 'push-button'.

## 1.3. Contributions of this Thesis

This thesis presents the following contributions to the current state-of-the-art. The seven previously published papers are listed below:

(i) Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, Sabine Rittmann, and Martin Wechs. COLA—The Component Language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München. September 2007.

(ii) Stefan Kugele and Wolfgang Haberl. Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*, Las Vegas, Nevada, USA, July 2008.

(iii) Stefan Kugele, Wolfgang Haberl, Michael Tautschnig, and Martin Wechs. Optimizing automatic deployment using non-functional requirement annotations. In T. Margaria and B. Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, volume 17 of *Communications in Computer and Information Science*. Springer, 2008.

(iv) Wolfgang Haberl, Stefan Kugele, and Uwe Baumgarten. Reliable Operating Modes for Distributed Embedded Systems. In *Proceedings of the ICSE Work-*

*shop on Model-based Methodologies for Pervasive and Embedded Software*, volume 0, pages 11–21, Los Alamitos, CA, USA, May 2009. IEEE Computer Society.

(v) Wolfgang Haberl, Markus Herrmannsdoerfer, Stefan Kugele, Michael Tautschnig, and Martin Wechs. One click from model to reality, 2009. Accepted for presentation at SAASE '09: Symposium on Automotive/Avionics Systems Engineering.

(vi) Wolfgang Haberl, Markus Herrmannsdoerfer, Stefan Kugele, Michael Tautschnig, and Martin Wechs. Seamless model-driven development put into practice. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 18–32. Springer, October 2010.

(vii) Wolfgang Haberl, Stefan Kugele, and Uwe Baumgarten. Model-Based Generation of Fault-Tolerant Embedded Systems. In H. R. Arabnia and A. M. G. Solo, editors, *Proceedings of the 2010 International Conference on Embedded Systems and Applications, ESA 2010*, pages 136–142, Las Vegas, Nevada, USA, July 2010. CSREA Press.

**Seamless MDD Automotive Development.** As we will see in the following chapter, *seamless* model-driven development of software-intensive automotive systems is one important building block to improve the system quality and moreover the state of practise and state-of-the-art. This includes besides a unique modelling formalism also modelling along different levels of abstraction. The practical applicability is shown by means of the case study explained in Section 8.2 [86,87].

**Data- and Control-Flow-Driven Specification of Automotive Systems.** Together with my colleagues, the data- and control-flow language COLA core [130] has been envisioned and realised within the integrated engineering tool COLA-IDE [86,87].

**Quality Improvements through Model Analysis.** One of the major benefits when working with mathematically well-defined modelling formalisms, such as the COLA core modelling language, formal verification and model transformation are enabled, just to mention two of the benefits. The detection of requirements inconsistencies through model checking techniques is detailed on in Section 5.2, the transformation of the COLA core language into Coloured Petri nets (CPNs), as published in [106], is one way to employ the power of available modelling

and analysis tools such as the Coloured Petri nets tools (CPN tools). Moreover, engineers are given hints of possibly undesired non-deterministic system behaviours (cf. Section 5.3).

**Generation of Executable Requirements Specification Documents.** The COLA-IDE allows to generate requirements specification documents straight from the tool. All artefacts stored in the model repository can be part of the generated output, for example, formal or informal requirements, COLA textual and/or graphical syntax. The look and feel can be adopted according to corporate design guidelines using stylesheets. The structure of the generated documents has been integrated into the next generation tool chain of our industrial partner and their user requirements specification process. Moreover, due to the tight integration of the generator into the engineering tool, the generated output can be considered as a *semantically tangible requirements document* as it is dynamic. This means that contained modelling artefacts can be simulated.

**Optimised Automatic Deployment.** The presented deployment methods follow the idea of a *systems compiler* that, analogous to a compiler for programming languages, transforms source artefacts into an executable system under consideration of optimisation goals. Optimisation is performed during system allocation with respect to memory and CPU demands and during scheduling. Here a makespan optimisation is performed. In general, many non-functional requirements can be considered as well [88,129].

**Reliable Deployment of Operating Modes.** The COLA core modelling language supports the powerful concept of *operating modes*. They are used to decouple complete system states into independent modes of operation, such as 'start up', 'driving', and 'parking'. This enables separate modelling and analysis of particular modes, thus reducing the complexity apparent to developers. Furthermore, resources are saved on the execution platform. During deployment, operating modes have to be considered especially during schedule generation as pointed out in [88,128].

**Automatic Generation of Fault Tolerant Operating Modes.** As a consequent extension, [89] describes modifications to the COLA automotive deployment concept, which are capable to tolerate to a certain extent hardware failures such as ECU or cable brake downs.

**Evaluation through Case Studies.** The feasibility of the presented COLA automotive approach has been demonstrated with several case studies partly in a tight cooperation with the industrial collaborator.

## 1.4. Outline of this Thesis

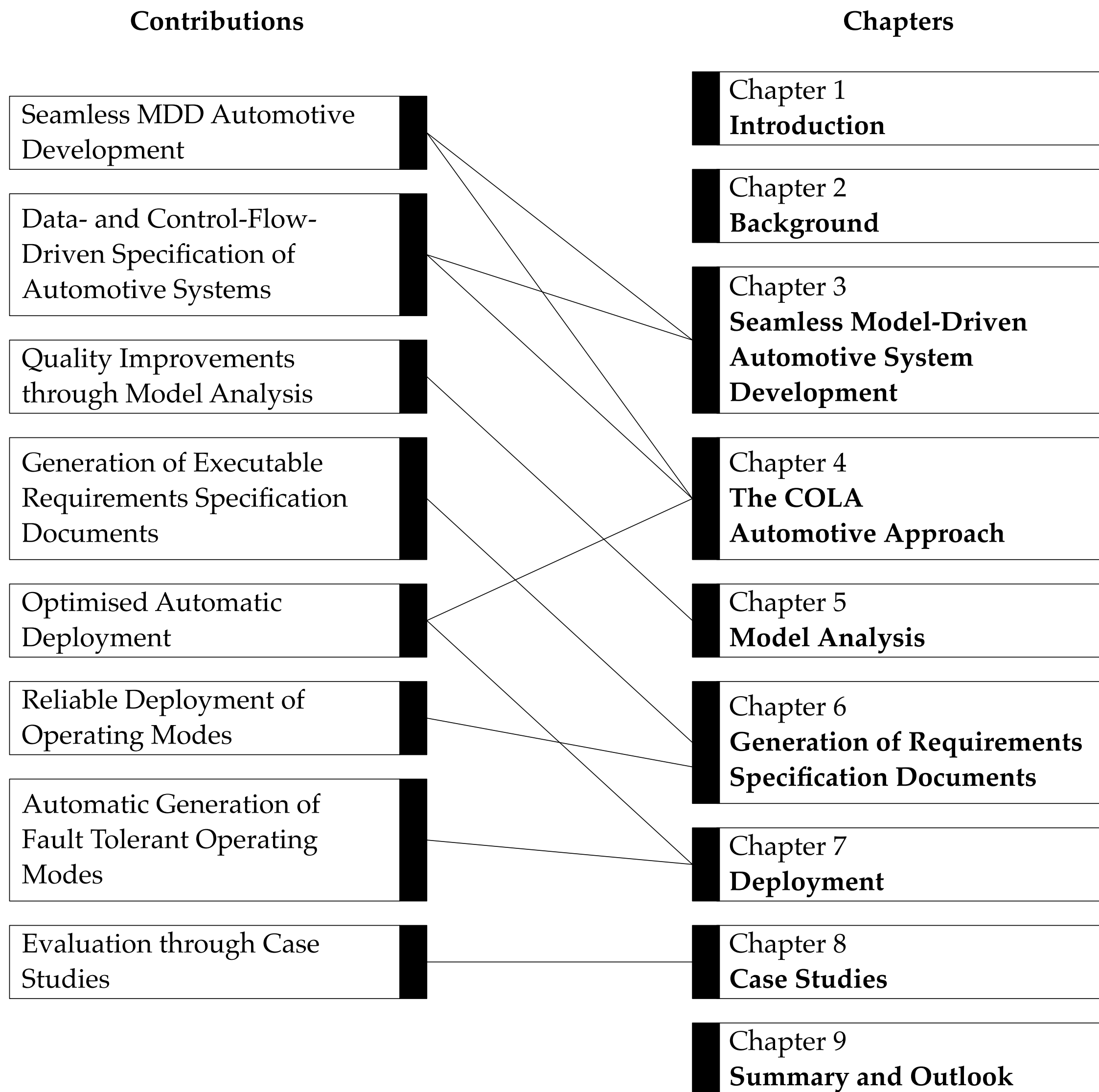



**Figure 1.5.:** Structure of this thesis.

In this dissertation, an automotive software development approach developed within a cooperation project together with BMW Group will be carried out. The

main idea is a *seamless* model-based development process of software-intensive automotive systems along different levels of abstraction. Figure 1.5 displays the contributions and chapters of this thesis.

Chapter 2 (*Background*) states basic facts about embedded systems in general, their omnipresent penetration of everyday's products, and their special requirements concerning timely execution. Next, a historical sketch of the automotive industry from purely mechanical to complex mechatronic-focused embedded systems is given. Their special characteristic and problems as well as future challenges is given next.

Chapter 3 (*Seamless Model-Driven Automotive System Development*) starts with deficiencies of commonly used engineering tools in Section 3.1 and based on this perception states the proposed solution in the following. The principle of 'separation of concerns' is seized on in Section 3.2 and the importance of having a well-defined modelling formalism is outlined in Section 3.3. Before summarising in Section 3.5, the step from isolated tools to an integrated authoring environment is described in Section 3.4.

Chapter 4 (*The* COLA *Automotive Approach*) delves into the developed COLA automotive approach with an introduction to the component language COLA, the way of modelling along different levels of abstraction, and the way of bridging the gap between a logical model and an executable system—the deployment process.

Chapter 5 (*Model Analysis*) points out that the well-defined foundation of COLA facilitates the application of formal methods. Besides COLA model analyses carried out in this chapter, also a translation into other modelling formalisms like Coloured Petri nets is possible.

Chapter 6 (*Generation of Requirements Specification Documents*) underlines the importance of requirements specification documents in the development of huge systems such as those included in automotive E/E architectures. They usually define the functional range of what to realise and not how to realise that particular function. Especially due to the tight cooperative work of OEMs and Tier 1 suppliers, a correct, consistent, and—if possible—complete specification is obligatory. This chapter presents an approach, as it is integrated into the COLA-IDE, to generate a well-structured requirements specification document directly from the model repository. A tight integration of the generated document,

the engineering tool, and the simulator makes the document dynamic.

Chapter 7 (*Deployment*) describes one of the main contributions of this thesis: an automatic deployment approach. The realised 'push-button' manner of the COLA automotive engineering tool allows an unattended transition from a logical (behavioural) model into executable code on the target platform. This process includes amongst others optimal allocation of software artefacts to hardware entities and the generation of a time-triggered schedule. All involved steps are performed automatically, which is a novelty in this field and a big step towards a *Systems Compiler*.

Chapter 8 (*Case Studies*) elaborates by means of different case studies the capabilities and feasibility of the COLA automotive approach.

Chapter 9 (*Summary and Outlook*) summarises this thesis and lists ideas of possible future research directions.

CHAPTER
# TWO

# Background

In the same way as the first automobiles were purely mechanical and did not contain any software or electric components at all, so automobility cannot be imagined without electronic advanced driver assistance systems, today. This transition from purely mechanical automobiles to mechatronic cars was immense. In the following Section 2.1, a short overview of embedded systems is given since the E/E parts of an automobile, which this thesis particularly deals with, build up a very complex networked distributed embedded system. Next, the mentioned transition and its cornerstones are outlined in Section 2.2, before characterising the particular aspects of the automotive domain in Section 2.3. This leads in Section 2.4 to challenges the automotive industry is faced with.

**Contents**


2.1. Embedded Systems: An Overview . . . . . . . . . . . . . . . . . 13
2.2. Historical Review . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.3. Automotive Industry Characteristics . . . . . . . . . . . . . . . 23
2.4. Current and Future Challenges . . . . . . . . . . . . . . . . . . . 33


## 2.1. Embedded Systems: An Overview

The importance and indispensability of embedded systems grew tremendously during the last four decades. One interacts with embedded systems in many situations in everyday life—be it wittingly or unwittingly. As their presence is ubiquitous, the necessity of quality, reliability, and safety grows. Their area of

application spans a very wide range of different domains with likewise different requirements and characteristics.

On the one hand, there are the more or less non-safety critical consumer electronic products like smart phones, MP3 players, or tablet PCs. Washing machines, microwaves, and home- or building automation are other examples where embedded systems are used to make our lives easier and more convenient. All mentioned examples have in common that their reliability requirements are circumstantial, whereas their primary design goal was usability including their **H**uman-**M**achine **I**nterface (HMI).

**human-machine interface**

On the other hand, there is a huge class of embedded systems posing special demands concerning reliability, robustness, safety, and timely execution. Their fields of application are manifold: plant automation systems, like for example nuclear power plants, airplanes and automobiles are smaller ones, but nevertheless big compared to small medical equipment like pacemaker, implantable cardioverter-defibrillators, or cochlear implants, just to mention a few. Any kind of failure in this category of embedded systems may be serious or even endanger human lives in the worst case. In any case, accident or maloperation, the operating company has to anticipate high warranty costs or claims for indemnification. It is remarkable that even the mentioned medical devices, which one could ascribe a very high quality, are not error-free: in the United States more than 200,000 pacemaker and implantable cardioverter-defibrillators were recalled due to software problems between 1990 and 2000 [144]. Engineers develop systems in these and many other application areas, where an imminent need for quality and absence of errors meets the problem's inherent complexity.

The importance of embedded systems also in the economic sense becomes apparent: according to Ebert and Jones [67], the worldwide market for embedded systems is around 160 billion euros with an annual growth of 9%. In fact, most microprocessors are used in embedded systems that are not first and foremost computers. Actually, up to 98% of all processors are integrated into embedded systems [28, 180].

In many cases, embedded systems are used to control complete systems or parts of them like an engine, a wing stabilisation, or certain parts of a plant. The term *control* refers to the fact that the system *reacts* on a continuously changing environment or user input with certain actions. An **A**daptive **C**ruise **C**ontrol (ACC) system, for instance, controls the speed of a vehicle in order to maintain a user-defined driving speed. Thereby the minimum distance to ahead driving cars is respected. Thus it reacts by controlling the pace, if the ahead driving car accelerates or breaks or, analogously, if the road condition, i. e., the environ-

**adaptive cruise control**

ment, changes. Such systems react continuously and are therefore executed in a loop. Consequently, Harel and Pnueli coined the phrase *reactive system* [93]. Figure 2.1 [179] depicts the control system 'Automobile' from a control-theoretic point of view. The system 'Automobile' is affected by the environment and by

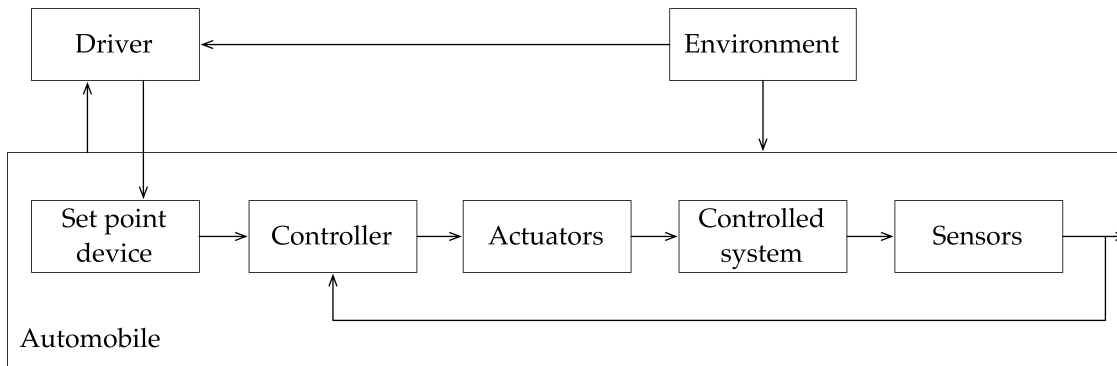



**Figure 2.1.:** Control system according to [179].

the user, who sets for instance the desired speed (set point device). The engine speed (actuator) is set accordingly by the controller. As a consequence, the speed of the car changes through the physical plant, which is detected by sensors and fed back to the controller for the next loop iteration. A common characteristic is the following: the physical world (environment) is measured using sensors and actions are triggered using actuators. As this thesis concentrates on the functional

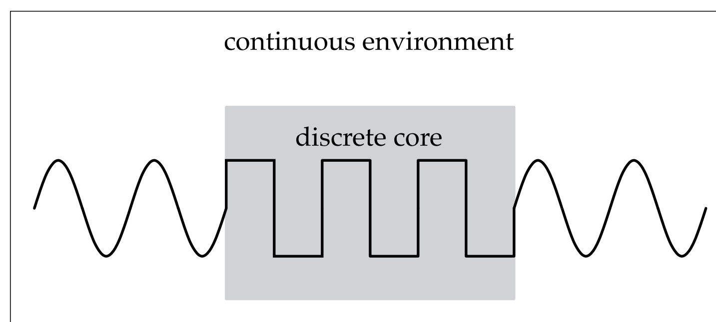



**Figure 2.2.:** Discrete core of COLA systems.

description, modelling, and deployment of automotive systems, it is the *discrete core* (cf. Figure 2.2) rather than its embedding into a continuous world, which is considered in the following. Therefore, the interface between them, i. e., the sensors and actuators and their analogue-digital (A/D) conversion and vice versa, is assumed to be given.

If not only the correct operation, but also the *point in time* and therefore a timely execution of a system is necessary for a safe, correct, and intended execution, such systems are referred to as *real-time systems*, as described by Kopetz [117]. Real-time systems are divided into two classes:

real-time system

(i) *soft* real-time and

(ii) *hard* real-time systems.

Soft real-time systems require that most of the results are computed within their deadlines. Results available after their demanded deadline reduce their usefulness and degrade the overall system quality. In contrast, hard real-time systems require *all* data to be available on time for a correct operation. Missing a deadline leads to a system failure. Hence, safety critical automotive functions, for example the airbag system or the electronic stability control are hard real-time systems, whereas, for instance, telematics or infotainment services are—if at all—realised as soft real-time systems.

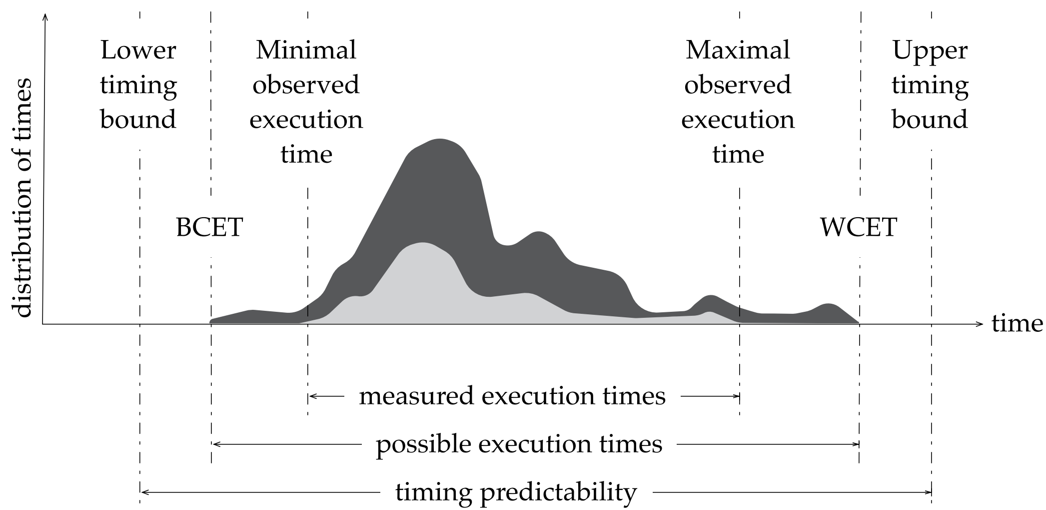



**Figure 2.3.:** Worst-case execution time according to Wilhelm et al. [198].

As time plays a crucial role for hard real-time systems, it is of importance to know their timing behaviour already at design time to guarantee that all deadlines are met. Therefore, the **W**orst-**C**ase **E**xecution **T**ime (WCET) (cf. Figure 2.3) of operations (on a specific target platform) is needed, i. e., the time required for the longest of all possible executions. In general, calculating the WCET is difficult, as all possible execution paths have to be checked, which in case of loops is impossible. Thus, in the automotive domain, the software development standard

worst-case execution time

for C programs, MISRA C [151], prohibits certain programming constructs such as recursive function calls (cf. the required rule 70 'Functions shall not call themselves, either directly or indirectly') whose runtime in many cases depends on the input values. In any case it is important that the estimated WCET is not less than the actual execution on the target, but is as close as possible. For a comprehensive survey of WCET analysis methods and tools, refer to Wilhelm et al. [198].

## 2.2. Historical Review

Development of automotive systems in general and automotive software in particular is best characterised in terms of a gradual evolution that took place over more than a century. In the following, the cornerstones of the transition from purely mechanical to electrical/electronic and software-controlled automobiles is outlined.

### 2.2.1. Classical Mechanical Engineering

First automobiles reflected the pioneering spirit of their constructors in the field of classical engineering disciplines, as they were purely mechanical and steam-powered. Later on internal combustion engines became the state-of-the art. At that time, cars were purely mechanical, without any electric device. In 1885 when Karl Benz built the first petrol-powered automobile—the Benz Patent-Motorwagen (cf. Figure 2.4)—, the pioneers at that time certainly could not imagine how an automobile would look like more than 125 years later. Neither their performance nor their functions besides the primary purpose—mobility and locomotion—could be brought to mind. Finally, the introduction of an assembly line between 1908 and 1915 for the Ford Model T, credited to Henry Ford, paved the way for a mass production. About ten years later, in 1926, the Robert Bosch GmbH invented a windscreen wiper, which was powered by an electric motor. This was one of the first examples of an electric device in a car.

In subsequent years, more and more electrical devices found their way into automobiles, the electric fuel injection marks one of those milestones. The late 70's and beginning 80's of the past century, however, can be thought of as the time when E/E components established themselves. Gradually, also the design objectives changed. In the beginning, for instance, engine controllers were optimised in terms of speed and power, whereas today the focus shifts towards energy efficiency. Over the last years the number of electronic components realising mostly safety

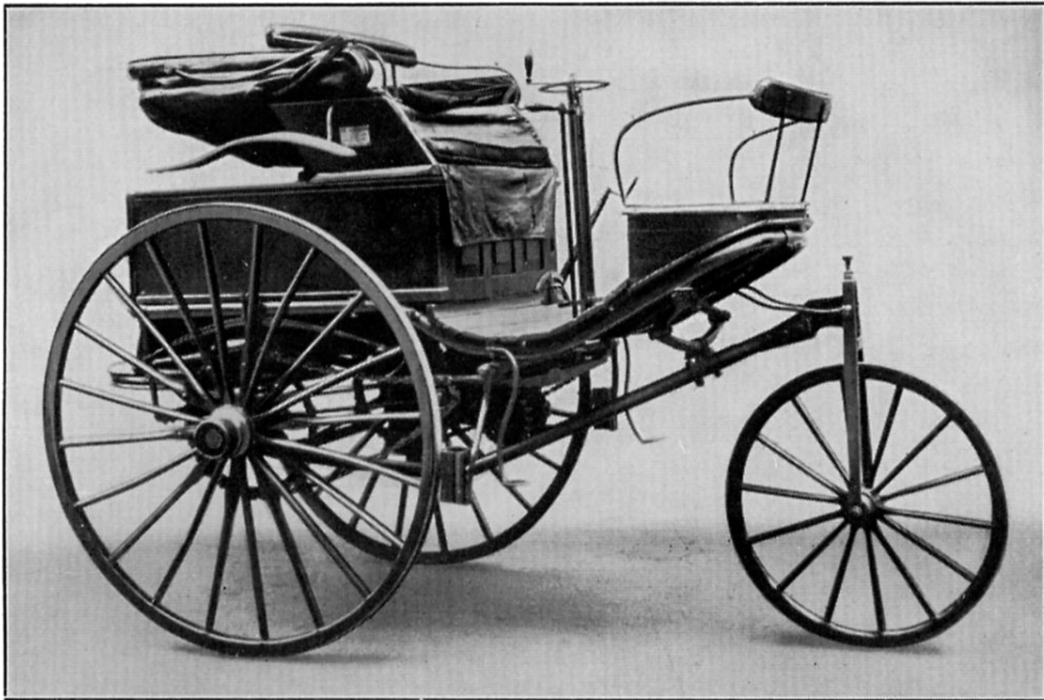


**Figure 2.4.:** The Benz Patent-Motorwagen Nr. 3 of 1888 [22]. Bertha Benz used it for the first long distance journey by automobile from Mannheim to Pforzheim, Germany.

and comfort functions increased significantly.

Nowadays, most of such electronic components run software affecting the desired features, or are realised as a combination of software and hardware. Today's premium class cars contain several thousands of software-controlled functions, that—in combination with electronics—facilitate about 90% of all automotive innovations. Figure 2.5 gives an excerpt from some E/E-related inventions in automotive history.

### 2.2.2. From Revolution to Disenchantment of Software

About 40 years ago, software began to change the automotive industry fundamentally [35]. At the early beginning of this turning point, only very few functions were controlled and realised by software. Broy [29] mentions the engine control and in particular the ignition system. Since then more and more electronic control units found their ways into automotive E/E architectures. Actually the number of assembled ECUs grew exponentially (cf. Section 2.4.3), which of course comes

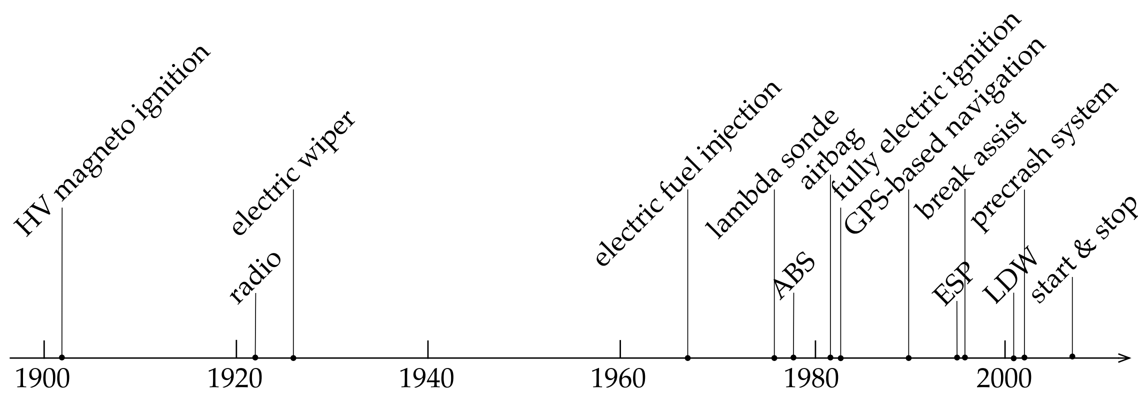



**Figure 2.5.:** Milestones in automotive E/E history.

along with an increase in the amount of software. Nowadays, all domains like *passive safety*, *powertrain*, *chassis/driver assistance*, *body electronic*, and especially the *human-machine interface* and *infotainment* domain are software-orientated.

At the early beginning, engineers wrote software parts of their systems under development predominantly using assembly language. In subsequent years and even till today, the most influential programming language for automotive software development has become C with all its advantages and disadvantages. During the last years, however, model-based development became more and more popular and turned out to be a promising methodology for software development for safety-critical embedded automotive systems. In the avionics domain, for instance, important European projects including the Airbus A340-600, A380, and functions developed by Eurocopter were realised employing the model-based paradigm [42]. There, the SCADE (**S**afety **C**ritical **A**pplication **D**evelopment **E**nvironment) tool-suite by Esterel Technologies [24] has been used for modelling, analysis, and DO-178B-level-A compliant code generation. The ASCET (**A**dvanced **S**imulation and **C**ontrol **E**ngineering **T**ool) products [69] by ETAS Group are other tools widely used in the automotive domain for safety-critical systems like ABS, ESC, and the engine control unit.

**DO-178B**

There were and are up to now good reasons to introduce more and more software-controlled functions in automotive E/E architectures. These are manifold and include amongst others:

(i) *Customers demand for more comfort.* Customers demand more and more functions that ease the usage of the automobile. Classical mechanical components are replaced by their electrical counterparts. This sounds simple, but many changes, yielding a tremendous increase in the overall complexity, are involved. Basically, the complexity can be divided into two parts:

(a) the E/E related part representing the physical components like the electric motors, the wires (including direct connections and busses), buttons, and electronic control units, and

(b) the system intrinsic complexity induced by relations to other components of the car.

A typical example is the window winder, which is replaced by its electrical counterpart—the power window. In each car door, a separate electric motor is used for automated lifting and lowering of the respective automobile window. Buttons, in contrast to hand-turned crank handles in the classical case, are used to interact with the system. Sufficient battery voltage is necessary for proper operation. The battery condition is provided via a bus signal. In addition, a function that was not necessary in the classical case becomes now necessary: the pinch protection, which reverses the moving direction as soon as it detects a resistance before being in the end position (closed). This simple example shows the problem: many E/E components communicate over busses in a distributed setting where complicated interactions have to be considered. The complexity is rooted in the functions' nature: highly complex software and hardware structures are realising a multi-functional behaviour in combination with their distributed and concurrent characteristics.

(ii) *Highly competitive mass market.* The automotive domain is a highly competitive mass market, where OEMs have to produce under excessive economic pressure.

In order to differentiate from their competitors and to maintain a certain corporate image, e. g. to be a driver of innovations, more and more functions are included into the system. As different manufacturers share a relatively high amount of same parts—as they are produced by the same supplier—it is software, which in many cases makes the difference and thus supports a corporate's image.

(iii) Last but not least, a plethora of new functions realised by hardware, software, or a combination of both is due to legal regulations. These are mainly related to environmental and security regulations, e. g. the *Directive 2005/66/EC relating to the use of frontal protection systems on motor vehicles* [187]. This provision applies from 25th November 2006 to both new types of vehicles and new types of frontal protection systems as separate technical units. The idea behind this provision is to reduce the severity of injuries to pedestrians—in particular if the vehicles are driving with a reduced speed (under 40 km/h).

According to a study [133] charged by the *European Commission, Enterprise Directorate-General, Automotive Industry*, the European Union had to suffer the following causalities in 2004:

(a) Pedestrians *killed*: 9,024

(b) Pedestrians *seriously injured*: 176,385

(c) Pedal cyclists *killed*: 3,418

(d) Pedal cyclists *seriously injured*: 115,224

As a consequence, some OEMs use pyrotechnic actuators to raise the bonnet in case of a pedestrian accident. This is done in a similar way to current air bag inflators. Pyrotechnic processes produce gas-inflating bellows, which in turn raise the bonnet providing more space between the engine block and the bonnet used as crush-collapsible zone.

Of course, for the realisation of this system new hardware components such as sensors, actuators, additional ECUs, and a number of wires—inducing again more complexity—are needed.

Besides all the undoubted advantages software and electronics facilitated in terms of comfort, safety, and pollution control, also drawbacks have to be mentioned. Software enabled the development of new functions that were not possible in a solely mechanical realisation. As a consequence, at the beginning of this millennium, nearly all OEMs pushed the development of software and electronics controlled components at a pace that made manageability almost impossible. A multitude of new functions were thought of and being developed by engineers without having a detailed informatics or related education. Thus, many of those realisations had more an ad-hoc characteristic than being well-engineered. In consequence, the overall E/E architecture became more and more complex, which made later maintainability a difficult task.

Mechanical engineering is traditional the predominant discipline of the engineers working in the development of an OEM. For a long time, software was neither seen to be a challenging nor an important part of the overall development process. Established, and of course also proven development processes that were instantiated in times when cars were more or less purely mechanical devices, were no longer able to cope with software and E/E architecture development. Koscher et al. [123] reveal that it is unclear whether vehicle manufacturers considered the possibility of adversarial in their architectural design and thus saw the problem of potential threats. In fact, they demonstrate how to adversarial control many automotive functions, which also includes disregarding the driver's input.

Recently, Rouf et al. [173] reported security and privacy vulnerabilities of a tire pressure monitoring system used in a wireless in-car network.

These aspects can be seen as the reason for unpredictable and—for the developers as well as for the customers—annoying random errors. After an initial revolution of software in cars, disenchantment arose. Looking at the amount and structure of warranty costs in the automotive domain, a need for improvement is inevitable: billions of dollars are spent for warranty costs each year.

Considering the number and also the reason for car recalls in Germany from 1998 to 2009 [124], one notices a rapid increase. Figure 2.6 depicts both the recalls and also follow-up activities related to them. According to the ADAC (German

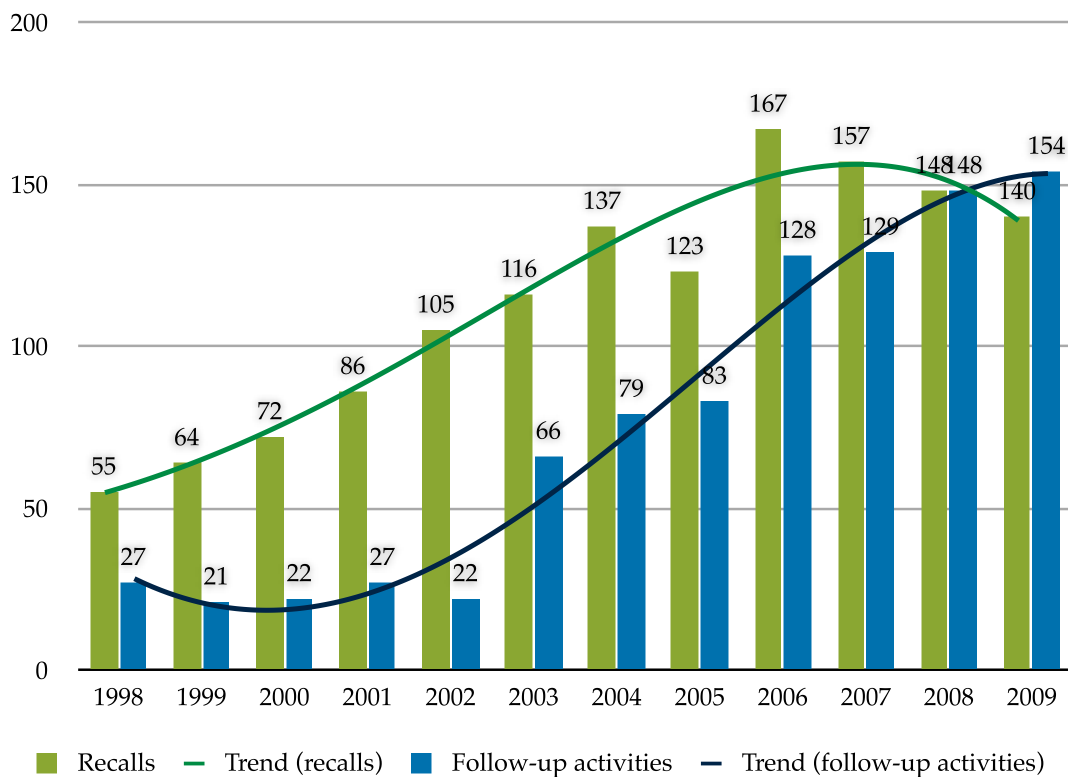



**Figure 2.6.:** Car recalls in Germany between 1998 and 2009 according to [124].

automobile club) [11], who analysed the reasons for car breakdowns in 2007, more than half of all breakdowns are dedicated to general electricity (including the battery) issues and the ignition system (cf. Figure 2.7a). In a retrospective, one can see that these components are exactly those whose breakdown contribution increased most within the period 1985-2007 (cf. Figure 2.7b).

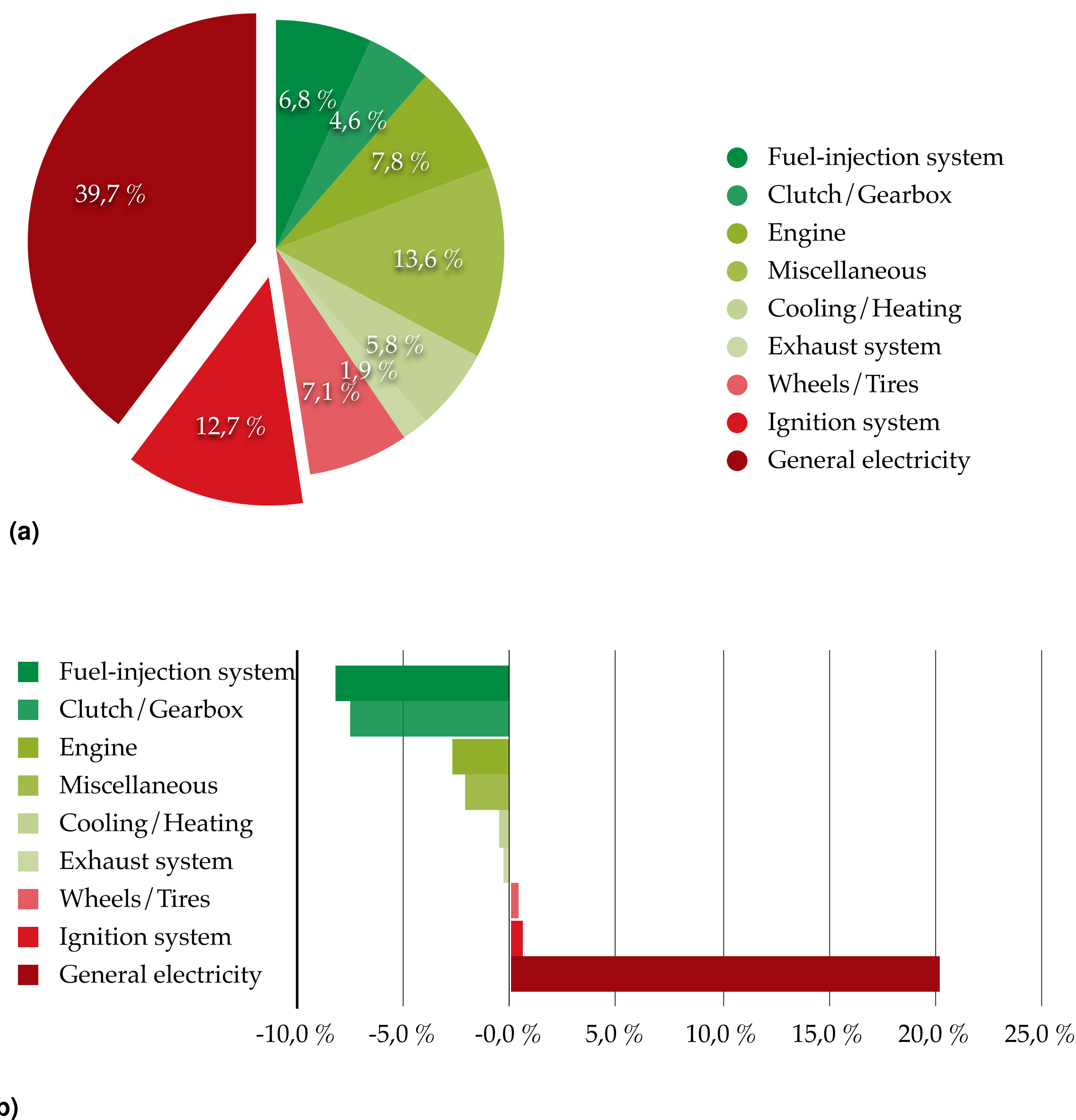



**Figure 2.7.:** (a) Car breakdowns in Germany in 2007 and (b) its evolution between 1985 and 2007 [11].

## 2.3. Automotive Industry Characteristics

The automotive industry is described using two different points of view: first, the economic perspective, which considers the company in a globalised world (cf. Section 2.3.1). Second, the internal perspective, which enlightens the main

artefact—the automobile itself—, is detailed in Section 2.3.2. This perspective observes the automobile from inside. Figure 2.8 visualises important attributes,

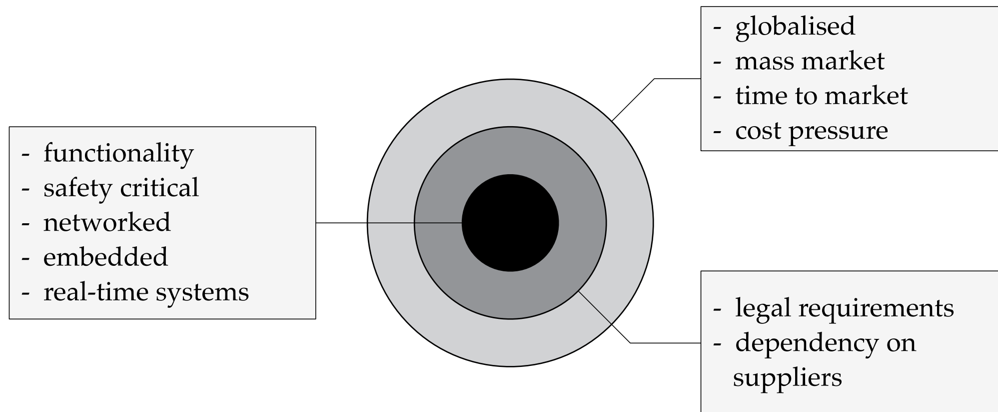



**Figure 2.8.:** Characteristic of the automotive domain.

which characterise the automotive industry. The automobile as central product is surrounded by constraints affecting the production. This in turn is embedded in an international context.

## 2.3.1. Automotive Industry

In the following, the automotive industry will be characterised with regard to some important aspects, which differentiates this specific sector of industry from for instance the avionics industry. In terms of safety, the avionics industry is no way inferior to the automotive industry—consider for example the avionics standard DO-178B and the upcoming DO-178C—but it cannot be considered as mass market, the supplier dependence is less marked and the drift to alternative propulsions has not happened yet.

### Mass Market

When looking at the automotive industry—in particular during the last four decades—one notices that it is subject to a fundamental change. Not only the revolution from mechanical to mechatronic systems has to be mentioned, but also a fundamental change in industry. More and more companies work in a highly competitive *mass market* and have to produce under excessive economic pressure.

> “Die weltweite Nachfrage nach Kraftfahrzeugen wird eine Million nicht überschreiten – allein schon aus Mangel an verfügbaren Chauffeuren.” (Gottlieb Daimler)

This becomes apparent when considering the last 110 years. The worldwide production of cars grew from only nine thousands in the year 1900 to more that sixty million in the year 2009 [12]. Figure 2.9 shows the impressive increase. Having these numbers in mind, one can smile about the quotation of Daimler,

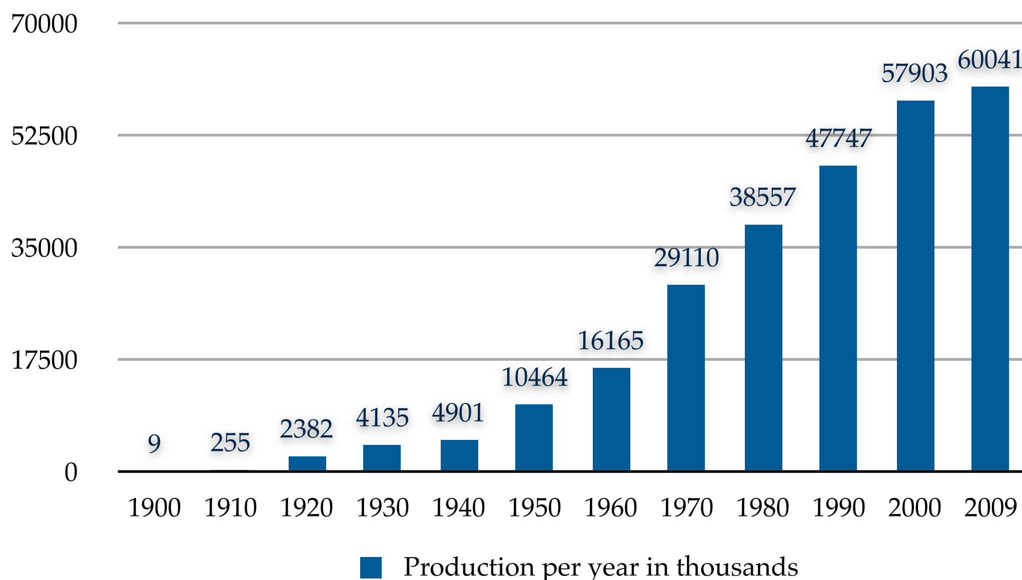



**Figure 2.9.:** Worldwide automobile production per year in thousands [12].

who says that the worldwide demand for automobiles will not exceed one million due to the mere fact of the lack of chauffeurs.

As cars are sold in the magnitude of millions per OEM and year, possible savings have to be elaborated in detail. A saving of less than EUR 1 per ECU might turn the balance to change software and use the cheaper ECU. Vendors from emerging nations, as for instance China and India, are just spreading their low-cost products to western countries. Of course, they cannot compete with quality and functional range yet, but they can compete with the price, which is for many people the most substantial argument. Hence, the pressure for an ever faster time-to-market of new models and derivatives grows notably.

### Functional Differentiation

Some of the already in Section 2.2.2 mentioned drivers for new functions can be used to describe the automotive industry. Different vendors use in many cases similar components sold by the same specialised suppliers, i. e., cars from different

OEMs have a high ratio of common parts. This is true for both software and hardware. It is not unusual that suppliers develop about thousand software variants for different OEMs with less that 10% of functional difference. In consequence, a corporate branding in design, sound and smell, and driving characteristics can primarily establish differentiation. Moreover, specially developed functions that are solely used by a typically small number of OEMs—in the best case a single OEM—make the difference. Typically the differentiation not by *design* but by *function* can only be done in the high price luxury segment, at least at the release of a new function. One example is the contact-free opening of the hatchback, which was introduced in Q4 of 2011 by the three big German OEMs Audi, Volkswagen, and BMW Group. Information about a case study with similar functional range is given in Section 8.3.

#### Dependency on Suppliers

As the automotive domain is a classical engineering discipline, which is rooted in its historical evolution, companies usually do not have special experience in the development of electronic or electric components. Oftentimes there is even no motivation to close this leak of knowledge. This fact however led to a state of dependence between the OEM and its suppliers. The OEM is dependent on the components' quality delivered by the supplier. The automotive supply chains include different stakeholders. First of all, there is the car manufacturer—or OEM—itself. Usually, OEMs provide the final product to their customers. Nowadays, many components of a car are not produced and developed by the OEM itself, but by a Tier 1 supplier. Mostly this includes the component's development and production on basis of a specification document provided by the OEM. It defines the general requirements of the product. Typical Tier 1 suppliers are Bosch, Delphi, Continental, and Siemens VDO. Finally, Tier 2 suppliers like for example Freescale or Infineon provide CPUs, memory, etc., which are in turn used in ECUs from Tier 1 suppliers. As the supplier's business model is to sell more or less similar hardware components to different OEMs, it is usually not possible for the OEM to get full access to the source code of the delivered component. The OEM has to trust that it does what it is intended to do. However, deeper insights are necessary to understand in more detail what is going on inside the component and hence in the interplay with the rest of the overall system. Today, the supply chain contains hundreds of companies that design and develop components for the OEM or a Tier 1 supplier [165].

### Legal Requirements

Passenger cars sold in the EU have to comply with the *Framework Directive for Whole Vehicle Type-Approval* [10]. Thus, automakers must follow more than 80 EU directives and regulations and—in the international context—even a larger number.

To some degree, this can be seen as a vicious circle. New functions as the LDW (**L**ane **D**eparture **W**arning) system help to reduce accidents. These in turn result in less often head-on collisions. In fact, those systems lead more often to impacts where only a part of the front is affected. Thus, more or less the same impact forces have to be absorbed by a smaller fraction of the car. As a consequence, regulatory authorities in turn adopted the legal requirements.

### Drift to Hybrid and Electromobility

During the last years, more and more automotive companies developed and introduced cars with alternative drive concepts. The most dominant ones are hybrid, purely electric cars, or the latter one with so-called range extender. Hybrid cars combine a combustion engine with an electric motor. Depending on the charging condition of the batteries, the driving mode (accelerating, breaking, etc.), and the speed, different scenarios (modes of operation) are possible:

(i) Only the electric motor is used. Especially in cities at lower speeds, when accelerating at traffic lights and in traffic jams, this mode is favoured. Note, that the cruising range is limited by the batteries' capacity.

(ii) Both, the combustion engine and the electric motor are used. This is favoured in sports cars, where a performance boost at lower emission rates is desired.

(iii) The shortcomings with respect to the distance in the first mentioned scenario can be avoided using so-called range extender. In case of low battery capacity, a special combustion engine is started to generate power for the electric motor.

Besides hybrid cars, electric cars only have either a single or numerous electric motors in their powertrain. Almost all of them are based on conventional combustion engine cars with respect to chassis, bodywork, and interior equipment. Generally, customers only waive energy consumers as for instance the seat heater. Here, the need for a detailed and rigorous consideration of operating modes in cars becomes obvious. Unfortunately, in the past operating modes have not been paid attention to in an appropriate way.

In the near future electric cars developed totally from scratch will be announced. This reinvention of the car itself was necessary to cope with the totally new future challenges. Putting several hundred kilograms of batteries into a traditional bodywork will most likely lead into a deadlock: the higher the weight of an automobile, the less the expected cruising range. Besides the weight of the batteries another problem is packaging, i. e., where to place the batteries best in the available construction space.

### 2.3.2. Automotive Domains

When having a closer look onto different components of modern cars, one notices at a first glance that it consists of dozens of embedded devices: the navigation system, the radio, the engine control system, and the airbag system—just to mention a few. But then, it turns out that all these embedded systems have completely different requirements in terms of reliability, correctness, response time, and a lot of other criteria. Therefore, they are coarsely classified into the following domains: *infotainment*, *body electronic*, *chassis/driver assistance*, *powertrain*, and *passive safety*, depicted in Figure 2.10.

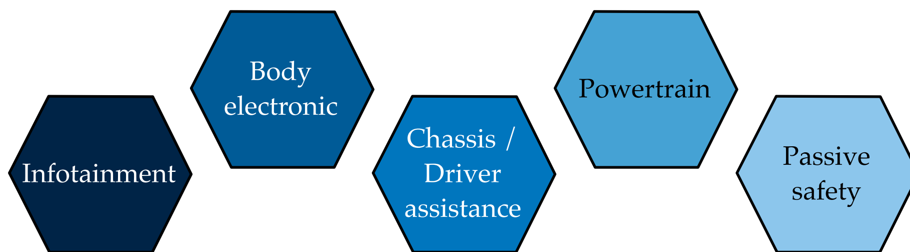



**Figure 2.10.:** Typical automotive domains.

Each domain poses specific requirements: infotainment components, for instance, settle for low reliability requirements, whereas passive safety components like the airbag control unit pose maximal requirements concerning reliability. The different categories will be explained in the following. As an example, Table 2.1 provides some reference points of the 2009 BMW 7 series premium class car (according to Weber [195]).

#### Infotainment

The term 'infotainment' subsumes the areas information, entertainment, and communication, which enter more and more modern automobiles. According to

| | Infotainment | Body electronic | Chassis/Driver assistance | Powertrain | Passive safety |
|---|---|---|---|---|---|
| Program size | 100 MB | 2,5 MB | 4,5 MB | 2 MB | 1,5 MB |
| ECUs (standard) | 4 | 14 | 6 | 3 | 11 |
| ECUs (options) | 12 | 30 | 10 | 6 | 12 |
| Bus type | MOST | K-CAN | F-CAN/ PT-CAN | LoCan/PT-CAN | SI-BUS |
| Bandwidth | 22 MBit/s | 100 KBit/s | 500 KBit/s | 500 KBit/s | 10 MBit/s |
| Messages | 660 | 300 | 180 | 36 | 20 |
| Cycle time | 20 ms - 5 s | 50 ms - 2 s | 10 ms -1 s | 10 ms - 10 s | 50 ms |
| Reliability requirements | Low | High/Low | High | High | Very high |
| Transmission medium | Optic fibre | Copper cable | Copper cable | Copper cable | Optic fibre |
| Bus topology | Ring | Tree | Tree | Tree | Star |

**Table 2.1.:** Characteristics of different automotive domains. The example shows values from the 2009 BMW 7 series, according to Weber [195]

their fields of application, infotainment systems pose high demands concerning data throughput and usability. Reliability demands in turn are relatively low.

Today, the driver has access to a multitude of information sources: beginning with traffic control messages and GPS, on-board diagnostic and status systems, going to full Internet access. Relatively new customer functions are realised using camera systems. The night vision system provides an infrared video stream of the vehicle's trajectory in order to warn the driver in the dark when pedestrians or animals are close to the road. A second example is a rear view camera used for reversing. All those applications have in common that they require extremely high transmission rates. Processing this amount of data needs relatively high computing power and a powerful transmission medium. Hence, typical bus types/protocols include the **M**edia **O**riented **S**ystems **T**ransport (MOST) [153] bus and Ethernet are used. As positive by-product of having camera systems in the car are additional information sources like road sign recognition (e. g. maximum permitted speed, 'when wet', or '10pm - 6am'). Besides information, entertainment is also a very important scope of infotainment. The classical FM radio found its way quite early into cars. In fact it was one of the first E/E function in cars. Of course, since then it has been improved concerning satellite reception, digital broadcast (DAB), MP3 and DVD playback capabilities. A well designed and ergonomic HMI (**H**uman **M**achine **I**nterface) is a precondition to interact with these systems without getting sidetracked. This, of course, holds for communication devices like mobile phones that have to be integrated and used in a hands-free way.

MOST

### Body Electronic

The body electronics domain includes functions that are usually not characterised as highly safety-critical. Nevertheless, functions with real-time constraints occur, too. This domain encapsulates functions that are not used to control the driving dynamics of a car. They appear in the chassis, or powertrain domain. Wipers, outside and inside lights, windows, doors and hatches, and seats are classical components of cars being controlled by electronics and software today belonging to the body domain. Figure 2.11 shows a fictitious but nonetheless realistic CAN [4–9]/LIN [139] (**C**ontroller **A**rea **N**etwork, **L**ocal **I**nterconnect **N**etwork) topology consisting of some seat modules. Each module (this example assumes a car with four seats) is responsible to control the following features (excerpt):

CAN
LIN

(i) Each seat can be moved forward and backward.

(ii) The position of each seat can be raised and lowered.

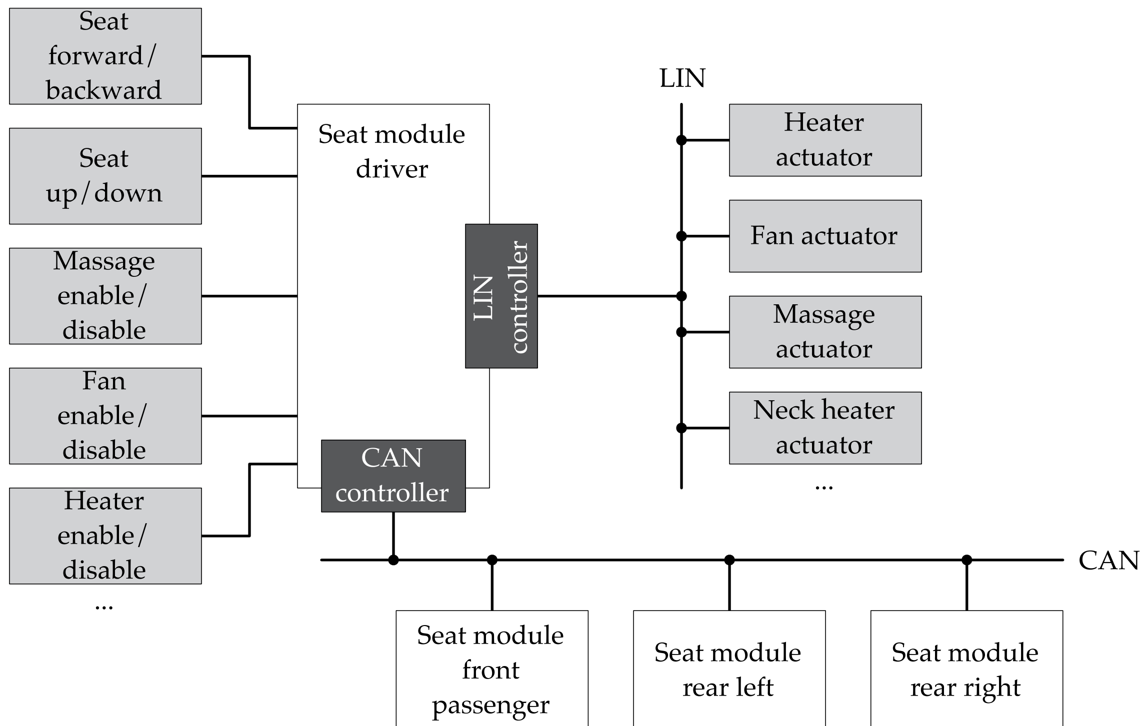



**Figure 2.11.:** Example of a seat module consisting of a couple of sensors and actuators, ECUs, and networks referring to [154].

(iii) Each seat can be heated.

(iv) Each seat comes with a massage function.

(v) Each seat comes with a fan.

(vi) Each seat comes with a neck heater.

All seat modules are connected to a CAN bus. Usually low-cost LIN-slaves connected to a single LIN-master are employed to activate the respective actors (heaters, fans, electric motors, etc.). In the depicted example, sensors are directly connected to the driver seat module ECU. A further connection via LIN bus is also possible in this setting.

From a functional point of view, most functions present in the domain are relatively manageable in terms of functional complexity. Here, the complexity arises from a very high degree of interconnectedness in the vehicle's wiring system.

### Chassis / Driver assistance

The chassis/driver assistance domain poses high demands with respect to hard real-time constraints. Besides the timely availability of input and output signals, the strict availability of contained functions is crucial from a safety point of view. As this set of functions can be considered as the link between the vehicle and the road, the contained functional range spans from ABS (**A**nti-lock **B**raking **S**ystem), ESC (**E**lectronic **S**tability **C**ontrol), to driver assistance systems like the lane departure warning system. Inputs to these systems are from the driver as well as from the environment. The driver steers, accelerates, and breaks the vehicle. Processed information from the environment contains the road conditions, the trajectory, and information of each wheel. In case of the LDW system, also the lane has to be analysed usually based on video sensors. Most of those functions demand detailed knowledge in control theory from the engineers.

**electronic stability control**

Similar to the following domain—the powertrain domain—many input signals have to be processed. Hereto, values of sensors distributed all over the E/E architecture have to be sent via possibly different buses and gateways.

### Powertrain

The powertrain domain encompasses everything that is necessary to convert driving power into propulsion. This includes traditional combustion engines, purely electric motors, and a combination of both in a hybrid setting. Like in the chassis domain, complex control laws with varying sampling rates according to the motorcycle are used. Today, engine controllers are calibrated using thousands of parameters. Depending on throttle position, current temperature, position of the gas pedal, and also the exhaust pollution, the fuel injection is controlled. Due to the rough environmental conditions in the close proximity of the engine (e.g. vibrations, varying and high temperature, EMC, humidity, etc.) high requirements are posed concerning the packaging and robustness of the E/E devices. Typical powertrain ECUs work with 32-bit micro-controllers running at hundreds of MHz [177]. Even multi-core ECUs are used in the powertrain domain, for instance in some engine control units. Mössinger [152] states that a high-end engine control unit contains more than 2 MBytes of flash memory and about 300 thousand lines of code. A widely used controller is the Renesas Electronics (formerly NEC) V850 with all its variants for example the Px4 as dual core which is certified for safety critical systems (ASIL D, SIL3).

#### Passive Safety

Passive safety subsumes all functions in a vehicle that help to reduce the effects after a crash. Of course, especially the possible injuries of passengers, pedestrians, and cyclists should be minimised. Established functions are for instance seat belts, airbags, and rollover bars in convertible cars. A new system, designed to protect outside traffic participants in case of a frontal crash, is a raisable bonnet. In its mode of action it is similar to an airbag. All the mentioned systems have in common that they pose hard real-time constraints. Hence, it's no wonder that for new airbag systems the highest **A**utomotive **S**afety **I**ntegrity **L**evel (ASIL) D with a tolerable hazard rate of $10^{-8}$ per hour is demanded according to the new ISO 26262 [104]. Consequentially, for those systems the FlexRay [75] and the **T**ime-**T**riggered CAN bus TT CAN [6] are used in many cases.

**ASIL**

**ISO 26262**

**FlexRay**

**TT CAN**

## 2.4. Current and Future Challenges

Derived from the status quo and the basic characteristics given in Section 2.3, some major challenges the automotive industry is faced with can be given.

### 2.4.1. Heterogeneity

As outlined above, each automotive domain poses specific demands concerning reliability, robustness, real-time capabilities, and environmental compatibility (e. g. temperature, EMC, vibrations, humidity, etc.). This list can be continued with aspects of more technological nature: we find in premium class cars a combination of different bus technologies like CAN, LIN, FlexRay, MOST, and Ethernet. Moreover, the operating system of different ECUs may vary. In many cases functions are realised in combination of hardware and software parts. This tight integration makes it costly to correct defects late in the development process. Besides technological challenges during the development process of automobile E/E systems, also organisational barriers have to be broken down. Dozens of Tier 1 suppliers have to be synchronised, which in turn have sub-suppliers in order to meet all deadlines. Heterogeneity becomes apparent in

(i) the used bus technologies,

(ii) the assembled electronic control units with,

(iii) the different operating systems running a combination of event- and time-triggered tasks,

(iv) the different suppliers working hand-in-hand with the OEM, which in turn synchronises interdepartmental work.

AUTOSAR

In the past, some OEMs wanted their suppliers to integrate their own software core consisting of operating system and basic software services. Nowadays, more and more OEMs proceed to demand AUTOSAR (**AUT**omotive **O**pen **S**ystem **AR**chitecture) compatibility including an AUTOSAR conform operating system running on the respective ECUs. This, of course, is a big step in the right direction towards reaching more homogeneity in automotive E/E architectures—at least homogeneity of technological nature. AUTOSAR follows the goals *modularity*,

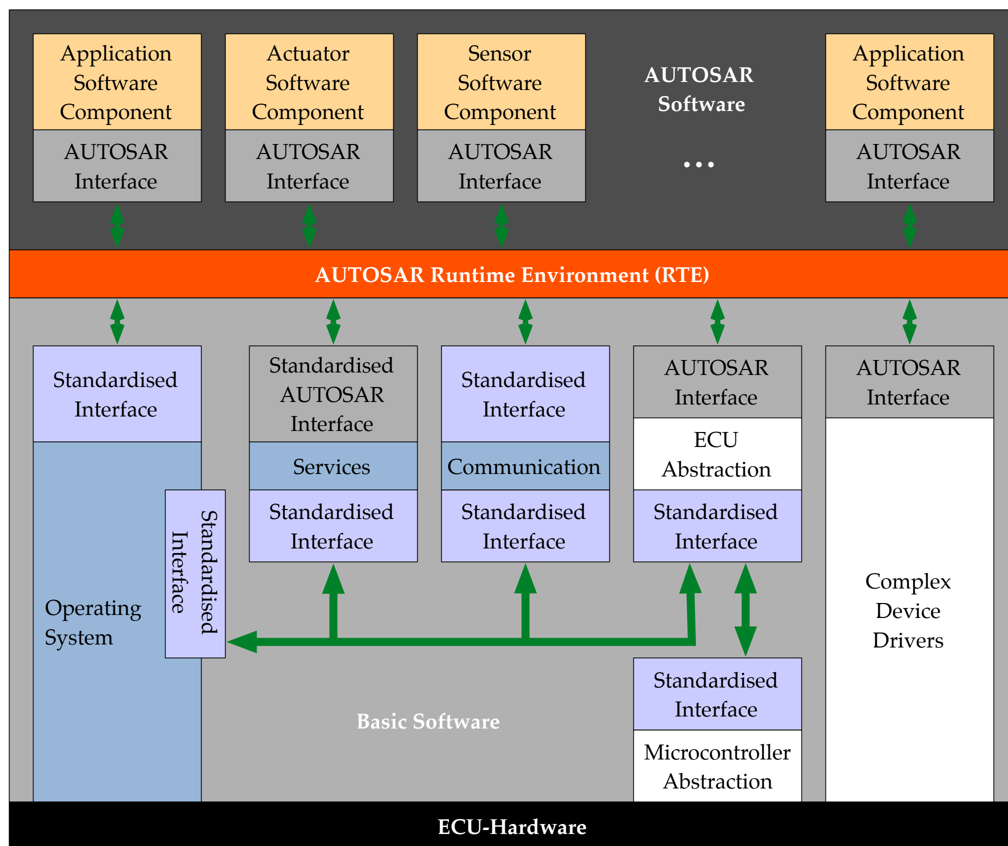



**Figure 2.12.:** Different AUTOSAR layers [1].

*scalability*, *transferability*, and *reusability* of functions by providing a common software infrastructure with the clear aim to be suitable for all automotive domains. For all layers depicted in Figure 2.12, AUTOSAR provides standardised interfaces facilitating the targeted objectives.

GENIVI Alliance

A second attempt to mention in this context is the GENIVI Alliance. This non-

profit consortium consists of major OEMs like BMW Group, GM, and PSA Peugeot Citroën, semiconductor and device manufacturer, as well as software companies. Their goal—as the 'IVI', which stands for **I**n-**V**ehicle **I**nfotainment, indicates—is to provide a range of compliance statements, and a compliance programme for GENIVI certification for automotive infotainment systems. The idea is to provide a Linux-based open source reference platform including operating system and middleware.

### 2.4.2. Clash of Cultures

According to Broy et al. [35], more than half of all replaced ECUs are technically error-free. There are sporadic errors, which are very difficult to reproduce and thus to isolate. Millions of euros are spent to replace *error-free* ECUs per year. One problem is that monitoring E/E components at run-time was not of major interest during the last decade, since resources were and are still limited. In consequence, not all errors occurring in the field are logged. Hence, only the entries in the error memory are available for diagnosis purposes. The root of failures is however not the limited error memory, but a badly accomplished automotive software or systems engineering. In practice, E/E architectures are not re-developed from one generation to the next generation of automobiles. On the contrary, 'proven' architectures are extended due to time and cost pressure. Oftentimes, E/E architectures are a result of structures that have evolved over time and—as experience has shown—are very hard to maintain. As automotive software engineering is almost never a green field approach, the importance of an extendable, well-understood and documented E/E architecture becomes apparent. In practice, however, companies have to struggle with legacy issues. On average, about seven years elapse between automotive generations with a so-called 'facelift' in between. Technological achievement develops at a much faster pace. Regarded from this vantage point, a two generations old automotive architecture can be considered as outdated. However, parts of these legacy architectures are still used today.

Former architectures were designed having maximum resource and cost efficiency as *the* most important design objectives in mind, current trends shift more towards maintainability, extendibility, predictability, and functional safety.

### 2.4.3. Control the Complexity

With the ever-growing number of electronic control units and software-controlled functions, the overall system complexity grew exponentially during the last years. The overwhelming increase in the number of ECUs becomes apparent in the following: whereas the Daimler-Benz W220 S-Class luxury sedan of 2003 had approximately 50 controllers [83], forthcoming luxury class cars will have up to 110 ECUs. Hence, we noticed a doubling in less than ten years. The given example is no particular case, but rather describes the evolution of E/E architectures of the whole branch of industry. In Figure 2.13, the progress over the last 25 years is depicted with a log-lin graph. Reichart and Heinecke [167] state that the number of ECUs has reached a limit due to packaging reasons and propose to concentrate the functional range on less but more powerful ECUs. This centralisation, in turn, poses increasing requirements concerning reliability and availability, and may cause pinning problems, i. e., the physical interface for connectors is limited.

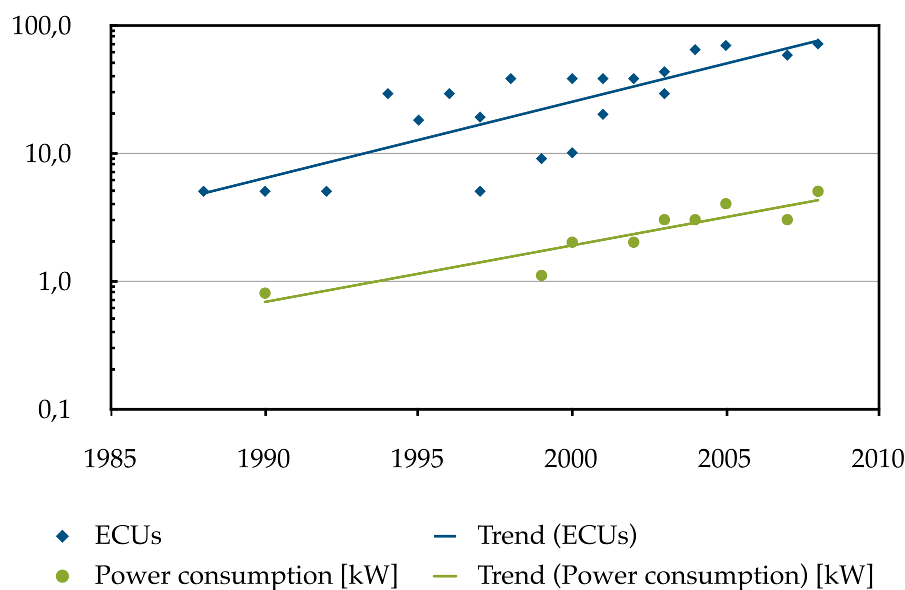



**Figure 2.13.:** Progress of the number of ECUs (according to Grimm [83]).

Besides the sheer combinatorial complexity induced by the high number of interconnected electronic control units and software functions, another source for system complexity is the large number of mostly subtle and in many cases also unexpected interaction patterns amongst the components (hardware as well as software). Today, more that 3000 functions are realised by software running on

ECUs, intelligent sensors and actuators. These functions are usually not realised on a dedicated ECU, but perform their service in a highly interconnected distributed networked system. According to Broy et al. [36] up to 100,000,000 lines of code (LOC) are realising software-intensive functions in an automobile, which is by far more than in the Space Shuttle or modern operating systems. Ebert and Jones [67] cite that about 1 GiB of software can be found in modern automobiles.

The introduction of automotive bus systems had two sorts of effects: on the one hand, numerous harnesses and cables could be economised, which in turn reduced production costs and the weight. As the weight is a significant factor for the $CO_2$ footprint, less weight is always desirable. Moreover, a reduced number of connection points saves time and cost during assembly and also reduces points of failures. On the other hand, however, the coordination of distributed real-time functions becomes more and more difficult: buses are shared by many bus subscribers, thus depending on the used bus, a bus schedule has to be generated, engineers have to think about arbitration and jitter. Due to sometimes used 'evolutionary ad-hoc' E/E architectures and topologies it is often hard to predict what happens when a new bus subscriber, for instance a new ECU, is connected to a particular bus. Different bus types are connected via gateways whose transmission times have to be known and also considered in order to guarantee end-to-end deadlines for time-critical system functions.

As mentioned above, several thousand functions in a vehicle are today realised by software. Not all of them are directly tangible customer features, i. e., features that can be noticed or used by a customer. However, there remains a multitude of features influencing each other in an often subtle and at the first sight unexpected way. These so-called *feature interactions* are besides the functional and the architectural complexity a further source of complexity. A classical example of feature interaction from the automotive domain is the power window. At least the following two features are involved, namely

**feature interaction**

**F1** : open/close the window and

**F2** : anti-trap protection.

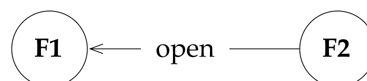



**Figure 2.14.:** Feature interaction example: if **F2** gets activated, the window is forced to open.

In Figure 2.14 a convenient graphical representation of this situation is depicted: if the anti-trap protection gets active, the window opens. For simplicity, this example does not consider the fact that the car's battery voltage has to be sufficient. More details on how to model the set of interacting features is given in Section 4.2.1. When considering requirements interdependencies in general, Carlshamre et al. [41] state that only approximately 20% of the requirements of any set of requirements are truly singular, i. e., they are neither related nor influenced by other requirements. For more information about requirements interdependencies, refer to Dahlsted and Persson [51].

### 2.4.4. Move from E/E Component-driven Development Towards Function- and Mode-driven Development

Up to now, automotive E/E development is highly hardware component-centric. That means, for each hardware component—usually an ECU—software parts are defined that are placed onto that particular ECU. This is due to the common practices suppliers (Tier 1) are integrated into the development process. In many cases a supplier is responsible for a defined ECU, which integrates different parts of different functions. In the future, especially in the context of AUTOSAR, OEMs will be enabled to exchange software components (SWC) from different vendors for instance due to economic considerations. The principle to use **C**omponents **O**ff-**T**he-**S**helf (COTS) mainly associated to hardware components is extended to software components, too. This shifts the focus away from the E/E component towards a function realised as a SWC, which in turn is then deployed onto an E/E component like an ECU, a sensor, or an actuator. Concerning functional safety, AUTOSAR provides features to achieve this aim similarly to what is called a **S**afety **E**lement **o**ut **o**f **C**ontext (SEooC) in ISO/FDIS 26262 part 10. As these elements (software or hardware components) are developed without having an item at the time of development, safety requirements are replaced by assumptions.

SEooC

When taking up again the example of hybrid drive given above, a function may behave totally different in view of fully electric, combustion, or a mixed drive. A seat heater, for instance, may only be enabled in the mixed or the combustion mode. With regard to fully electric vehicles, however, the situation is different and so is the functional behaviour: one can imagine that the seat heater is only active when the car is decelerated and brake energy is recuperated. Thereby decelerative forces are used in a process known as recuperation, which uses the electric motor as a generator to produce energy and charge the batteries.

By means of this example it gets clear that—from a functional point of view—

even functions of a manageable size and complexity like a seat heater have to be adopted to fit into future automobiles with alternative powertrains.

CHAPTER

# THREE

## Seamless Model-Driven Automotive System Development

Model-driven development (MDD) is an emerging engineering paradigm, which—in many areas of the automotive and other domains as well—becomes a de-facto standard development approach. However, its potential is not fully exploited. This chapter outlines a methodology to go far beyond the state of practice. After motivating the approach in Section 3.1, the concept of abstraction and modularisation realised within this work will be presented in Section 3.2. The possibilities that present themselves, when building upon a proper theory, are enlightened in Section 3.3. Moreover, it facilitates the integration of hitherto ad-hoc built tool chains into a single integrated engineering tool, which will be discussed in Section 3.4.

**model-driven development**

### Contents


**3.1. Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**3.2. Separation of Concerns Through Abstraction and Modularisation** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**3.3. Theoretic Foundation: A Fertile Soil for Formal Methods** . . . 46

**3.4. From Isolated Tools to an Integrated Authoring Environment** 48

**3.5. Summary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

## 3.1. Introduction

At the early beginning, when electronics and software found its way in the automotive domain, the pioneering engineers were faced with tremendous challenges. Neither tools, nor development processes were established at that time. Many technological problems had to be understood and solved, which was successful in many cases. After roughly five to six generations of cars since the first electric and electronic components had been used, most technological problems are solved. As the automotive E/E systems' complexity grows drastically (cf. Figure 2.13), in fact exponentially in the number of ECUs and the functional scope, the bottleneck nowadays is no more of technological nature, but is rooted in inadequate engineering capabilities. According to Sangiovanni-Vincentelli and Di Natale [177], most modelling tools have amongst others the following problems:

**(P1)** Lack of separation between the functional and the architectural model.

**(P2)** Lack of support for defining the task and resource model.

**(P3)** Lack of modelling support for analysis and back-annotation of scheduling-related delays.

**(P4)** Lack of sufficient semantics preservation.

Later in this thesis, these four aspects are revisited and a solution is proposed.

## 3.2. Separation of Concerns Through Abstraction and Modularisation

It is likely that the increase in system complexity will continue in the future. Besides the functional and technological complexity, also the organisational challenges with different stakeholders have to be under control. In the design and development of modern cars an immense number of interdisciplinary experts are involved. No single engineer is able to be an expert in all contained disciplines. Still, when only the E/E field is considered, we observe the following participants:

(i) requirements engineers,

(ii) function specialists,

(iii) function architects,

(iv) experts for networked systems,

(v) user interface and HMI designers,

(vi) functional safety engineers

and many more. To enable modelling with a uniform engineering tool, each involved engineer should be able to work with artefacts only in those parts of the model s/he is particularly interested in and has the right to do so. The introduction of views hides the system's complexity apparent to developers and also controls access authorisation. This can be achieved by abstraction, or—to be exact—a hierarchy of different levels of abstraction. At the beginning of product development, other aspects are of relevance than later in the process. For example, the goal to develop an adaptive cruise control system is much more abstract than its technical realisation using for instance long range and short radar sensors, integrated into a special ECU and connected to particular bus systems.

The ideas to structure and modularise systems are not new, but go back to seminal works by Dijkstra [58] and Parnas [157] in 1972. Decoupling, structuring, and hierarchisation are some examples of simplifications in system design. But designers of complex, reliable, safety-critical, and networked embedded (automotive) systems still face complex problems. The task remains complex even though structuring techniques are used since the complexity is problem-inherent. However, what these methods *can* provide is to tame the complexity such that developers are able to manage the design process.

Tony Hoare got to the heart of the problem in his famous and groundbreaking Turing award lecture [98]:

> [T]here are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

This thesis proposes to use a hierarchy of levels of abstraction, leading to a separation of concerns as claimed by Dijkstra [59]. On each level, special artefacts are modelled that are of relevance in a particular process step. Consequently, at early stages modelling artefacts are more abstract than in later steps. For instance, functional requirements are usually quite abstract, as they do not consider a technical realisation, which in turn is quite concrete. Hence, to capture all relevant modelling artefacts, a hierarchy is constructed. This ordered hierarchy starts with a

very abstract level and finally ends in a very concrete and technical one. Figure 3.1 depicts an abstract example with $n$ levels of abstraction ($\mathscr{L}_1$ to $\mathscr{L}_n$). The transition

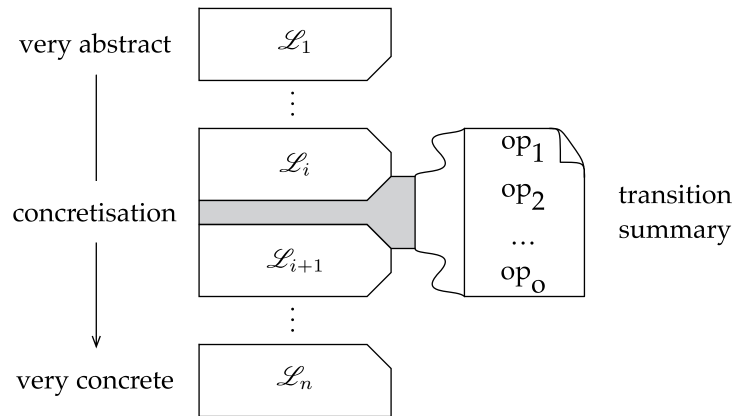



**Figure 3.1.:** Hierarchy of abstraction levels with transition summary.

between different levels of abstraction—usually from top to bottom—has to ensure the following criteria.

**C1** *Reversibility.* When transiting the gap between two levels, all performed actions have to be logged. Reading the log in inverse order gives an explanation for the existence of artefacts on lower levels. Reversibility is of importance to guarantee traceability from concrete levels back to more abstract ones (and vice versa), desired by regulatory authorities.

**C2** *Enrichment.* From a higher level of abstraction to a direct subsequent lower level, there has to be a real enrichment concerning concreteness, i. e., there must be a real increase of information. Otherwise the respective levels would not have been necessary and the two levels could be collapsed.

**C3** *Completeness.* Specifications on more abstract levels have to be realised in the lower levels.

One might ask why properties **C1** to **C3** are so important?

First of all (**C1**), when proceeding from an abstract level to a concreter one new artefacts are possibly generated or existing ones modified. In reality, however, it will not be possible to descend the hierarchy only using a single operation. Such a transition usually consists of a set of operations. A chronological record ($op_1$ to $op_o$) of all operations yields the transition's summary as depicted in Figure 3.1. Moreover, when reading the record in reverse order, traceability is facilitated.

For certification authorities, traceability is important. Thus, the reason for an artefact's existence has to be traced back to a certain requirement. This is demanded by maturity models like CMMI (**C**apability **M**aturity **M**odel **I**ntegration) [47] and ISO IEC SPICE (**S**oftware **P**rocess **I**mprovement and **C**apability **De**termination) [103]. Moreover certification standards like DO-178B (Software Considerations in Airborne Systems and Equipment Certification) [174] and ISO 26262 ('Road vehicles – Functional safety') [104] demand requirements traceability, as well. In the context of automotive software, the ISO/IEC 12207 and ISO/IEC 15504 have been adopted to fit the demands in the automotive domain. The most serious change in *Automotive SPICE* was *requirements traceability*. These adoptions were made by AUTOSIG, the **AUTO**motive **S**pecial **I**nterest **G**roup, in 2001. Therefore, for each artefact, traces can be given, which provide a summary for the existence of an artefact. According to Ramesh et al. [166], comprehensive traceability supports developing a better quality product, improve development and maintenance of the software, and possibly lowering the system life cycle cost. Figure 3.2 shows a 3-levelled architecture—as used later in this thesis—with a couple of linked artefacts between the levels. Concretisation and thus reduction of abstraction is visualised by following the directed arcs from top to bottom. For instance, the second rectangular artefact of the second level ($\mathscr{L}_{logical}$) is reached from two circular artefacts of the first level ($\mathscr{L}_{feature}$). Hence, the existence of the square is justified by the circles. This proceeds down to the lowest level of abstraction ($\mathscr{L}_{technical}$). By traversing the links back, i. e., upwards, a summary for the existence of an artefact is given. In the depicted example, the star in the middle of the lowest level has two justifying artefacts on the second level which in turn have one and two artefacts on the highest level of abstraction, respectively. The thick grey bottom-up arrows represent the back linking.

**Automotive SPICE**

**AUTOSIG**

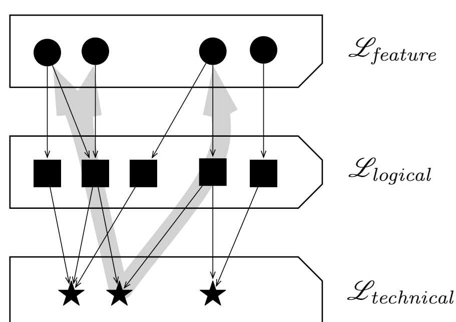



**Figure 3.2.:** Traceability links between artefacts.

Besides certification reasons, traceability also provides engineers with insights about which artefacts s/he has to review on lower levels in case of changes on higher levels. For further details about traceability in MDE, please refer to Galvao and Goknil [78].

Next (**C2**), from one level to the next more concrete level there is a real information enrichment. Without a real enrichment, the lower level would be obsolete and thus both levels could be merged. The goal, of course, is to give more and more details towards the final system under development. The allocation decision, for example, enriches a logical model with information where deployable entities should be executed.

Finally (**C3**), it shall be guaranteed that specifications on abstract levels are fulfilled on lower, more concrete levels. For example, a requirement has to be realised by a behavioural model, (ii) its semantics has to be preserved during code generation, (iii) the code executed on the final target platform, again, has to preserve what has been modelled and what was defined in the requirements.

## 3.3. Theoretic Foundation: A Fertile Soil for Formal Methods

Today's automotive software development is far away from being based on formal models and methods. Only some aspects during the automotive software development process are based on models and the model-based development paradigm. As not *all* software-related parts are modelled using the MDD method, the full capabilities are not exploited compared to what would be possible, if the process was based on both seamless *and* formal model-driven engineering. However, some tools working on semi-formal foundation like for example MATLAB/Simulink/Stateflow are applied in the daily practice. The widely used UML is not based on a rigorous, formal theory and is therefore unsuitable in its comprehensive and not specialised instantiation for safety-critical automotive systems.

Automatic C code generators are restricted to a subset of the modelling language, actually on the formal, well-understood, and deterministic parts. Besides the mentioned code generation also (model-based) test case generation (cf. also Utting et al. [190] for a taxonomy of model-based testing) and consistency checks are applied already today. Jackson [105] however objects that testing is not good enough, as exhaustive testing for systems of realistic size is intractable. Nevertheless, automatic test case generation from models or code as presented by Holzer et al. [99] is a very important brick in the tool box of systems engineers,

always having in mind Dijkstra's famous pronouncement [57] that testing is used to show the presence but not the absence of errors.

In order to upgrade the possibilities of having formal models, Kordon et al. [122] claim that one has to consider MDE as a generic framework in which verification plays an important role. Test case generation, verification, theorem proving, and type checking are only some ways to improve the quality of automotive software-intensive systems. D'Silva et al. [62] give a very good overview of automated techniques for formal software verification.

Moreover, formal models based on a comprehensive modelling theory enable model simulation—a sufficient level of detail assumed. According to Conrad and Doerr [48], executable models that can also serve as starting point for code generation can reduce software development time between 20-50%. The today's applicability of formal methods was not enabled by just a single technological and scientific achievement, but instead by advancements of different aspect. First, the available computing power, used to apply for instance time-consuming formal verification techniques, grew many times over the last decade. Second, there has been a great process especially in the field of efficient decision procedures like SAT (Boolean **SAT**isfiability of propositional logic) and SMT (**S**AT **M**odulo **T**heories) solvers, i. e., formulae consisting of a propositional part and parts of a different logic. Hence, Rushby [176] observes that formal methods are entering the mainstream. Of course, model checking of entire large systems, or exhaustive testing is still infeasible, but certain highly safety-critical sub-systems can be treated in particular. However, most success stories given in the literature are from the avionics rather than from the automotive domain. This may be rooted in the fact that rigorous engineering and formal models and methods entered the avionics development process quite early compared to the automotive field.

Summarising one can say that a theoretic foundation of the modelling theory has the following advantages:

(i) *Formal analysis and reasoning* can only be performed reasonably if models are based on a rigorous theory. Then, systematic verification and validation techniques can be applied in order to improve the system's quality. They include amongst others: model checking, theorem proving, simulation, test case generation, and consistency analysis. In Section 5.2, a technique is presented, which is capable to check in an early development stage consistency of formally specified requirements.

(ii) *Model transformations (model-to-model: M2M).* Whenever the use of a single repository building upon a formal product data model cannot be used

and a transformation between models (model-to-model transformation) becomes unavoidably necessary. Moreover, in the case that powerful analysis tools building on a different modelling formalisms are employed, model transformation is essential. This case is further exemplified by describing a translation schema from COLA to Coloured Petri nets in Section 5.4.

(iii) *Model-to-Code transformation (synthesis)*: Whenever source code or executable models are desired, they are synthesised. C code generation (cf. also Haberl et al. [90]), VHDL code used for hardware synthesis (cf. also Wang et al. [193]), and SystemC code generation [192] are only three examples within the COLA automotive approach.

As we will see in the following section, a well-defined modelling theory also bridges the gap between different product data models occurring in loose coupled engineering tools.

## 3.4. From Isolated Tools to an Integrated Authoring Environment

Today's use of tools, or better ad-hoc coupled pragmatic tool chains differ considerably from what can be considered as an *integrated* and *seamless* development approach. Before outlining the vision and solution in Section 3.4.2, the current state of practice is explained in the following section.

### 3.4.1. Daily Practice

At the present time, the modelling process can best be characterised as a pragmatic coupling of different modelling and engineering tools forming tool chains rather than having a single integrated authoring tool. The tool range spreads from general-purpose office tools like text and spreadsheet processing tools to specialised requirements engineering, behavioural modelling, or code generation tools. Some of them are quite good interoperable especially when they come from the same tool vendor. However, most of them are only coupled using 'gluing' adapters ('○—●' used as illustration in Figure 3.3) to build ad-hoc tool chains. In consequence we observe *integration gaps* between tools. Each tool ($t_1$ to $t_n$) works upon its own **P**roduct **D**ata **M**odel (PDM) ($PDM_1$ to $PDM_n$). This requires converting artefacts back and forth between tools as development proceeds. Still, there is neither a single tool capturing all steps needed during the automotive system

**product data model**

development, nor a seamless tool chain without integration gaps. The pragmatic approach has the following drawbacks:

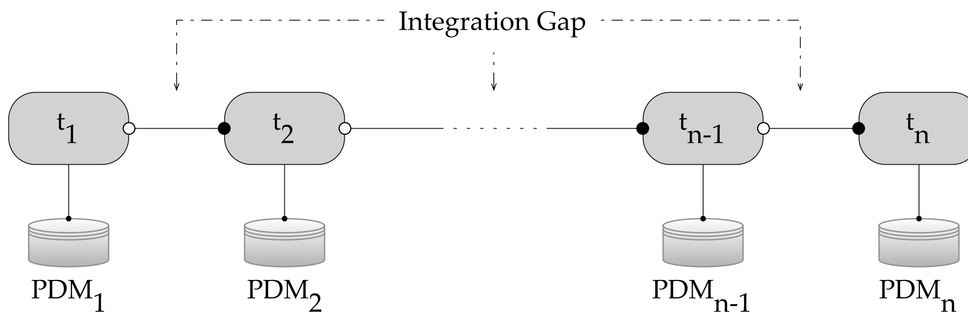



**Figure 3.3.:** Tool integration gap.

(i) The first and most visible difference between tools is the graphical user interface. From a technical point of view that seems to be negligible, however, psychology tells us that fears of using a tool in a wrong way, or being overwhelmed by the tool, does not humble the barrier between tool and engineer. On the contrary, it decreases acceptance amongst engineers. This fact could be studied during the work at industry partners trying to roll out new modelling tools. In the worst case, missing acceptance and thus lacking support by engineers could easily jeopardise the use of new tools.

(ii) An akin aspect coming along with the user interface is a subtle pitfall. Modelling is oftentimes performed using a graphical language like for example UML. Amongst others, UML supports modelling of automata using Statecharts. The concept of automaton to model state transitions of systems is common to other tools as well. MATLAB/Stateflow by The MathWorks, ASCET-SD by ETAS, or SCADE by Esterel Technologies support a state transition based modelling concept. However, the problem is that developers have an intuitive understanding of automaton and their semantics. Using different tools with similar graphical syntaxes, but (slightly) different semantics is very problematic.

(iii) The mentioned adapters are oftentimes realised using hand-written scripts trying to map artefacts from one tool to the other. This presupposes that the source and destination product data models are well-understood by the integrator developing the adapter. As experience teaches, this process is error-prone and not future-proof: each time, one of the tools changes—be

it the product data model itself or the tool interfaces—adapters have to be checked and if necessary have to be adjusted. Each modification, however, involves the risk of an unintentional insertion of errors. This scenario only works as long as the tool vendors document their changes. If no changes are documented, the user of the tool chain gets to know them only in case of a noticeable malfunction, which may be too late.

(iv) The mapping only works if an artefact in the source data model has a corresponding determination in the destination data model. At our industrial collaborators, this turned out to be a very pressing issue. If the destination data model does not have a semantically fitting concept/attribute, this information cannot be used in a useful way, yielding information loss.

(v) Besides modelling artefacts that have to be passed between tools, also the semantics amongst them has to be preserved. This point is possibly even more important than the mapping of all data. On the one hand, if not everything can be mapped, the engineer gets to know the information loss. On the other hand, if semantics cannot be preserved, the passed model becomes invalid, as one cannot trust in the preservation of its semantics.

(vi) Tool adapters may be a short term solution to pass modelling artefacts from tool to tool with all the already mentioned drawbacks. One very important issue, however, is not addressed: it is insufficient to *only* model artefacts. Rather, their *relationship* has to be captured, too. If related artefacts are modelled within different PDMs, it is hardly possible to manage their relationship, which is essential for traceability.

(vii) Likewise noteworthy is the number of possible adapters between tools. When having $n$ different tools, one needs in the worst-case quadratic many adapters, which—of course—is not maintainable.

In Table 3.1, commonly used tools during the development process—beginning from requirements engineering through to production code generation—are listed.

To overcome the mentioned problems, Broy et al. [33,35] pose the vision of an integrated modelling approach, which supports the creation, maintenance, and also the generation of new artefacts out of already modelled ones.

| | Notation | Tool |
|---|---|---|
| Requirements<br>(customer, system) | *natural language*<br>graphical notation (statecharts, activity diagrams, message sequence charts) | *requirements management tools*<br>Word, Excel, DOORS, UML tools |
| Design<br>System architecture | (formal) *models*<br>data-flow, control-flow, UML, SysML, AUTOSAR | *modelling tools*<br>MATLAB/Simulink/Stateflow, ASCET-SD, SCADE, Rhapsody, CATIA PLM V6, Siemens NX, Enterprise Architect, ... |
| Code | *formal languages*<br>Ada, C, C++, ... | *IDEs* and *code generators*<br>Eclipse, Netbeans, Visual Studio, TargetLink, ASCET, Simulink Coder |

**Table 3.1.:** Commonly used COTS tools during the product development process.

## 3.4.2. Solution

Besides the already discussed problems, issues of different nature arise, too. The multitude of used tools is not disjoint in terms of capabilities. That means, they offer partly similar functionalities. Hence, for a company it is crucial to have strict and mandatory rules 'what' to model 'when' in the process with 'which' tool. However, experience has shown that it is difficult to enforce and also to monitor modelling guidelines. Tool capabilities are used although they should not, especially by inexperienced engineers. As a result, the same or slightly different artefact is modelled unintentionally in different tools, yielding redundancy. Redundancy is very dangerous since it is not clear with which artefact to proceed in the subsequent process steps and if they are modelled intentionally, how to maintain consistency.

Consequently, the use of a *central repository* ('single point of truth') where all artefacts and their relations are stored has many advantages:

**single point of truth**

(i) There is no problem of redundant artefact storage—assumed the data model is designed accordingly.

(ii) The central storage of modelling artefacts facilitates systematic data analysis like for instance consistency checks. There is no need anymore to first

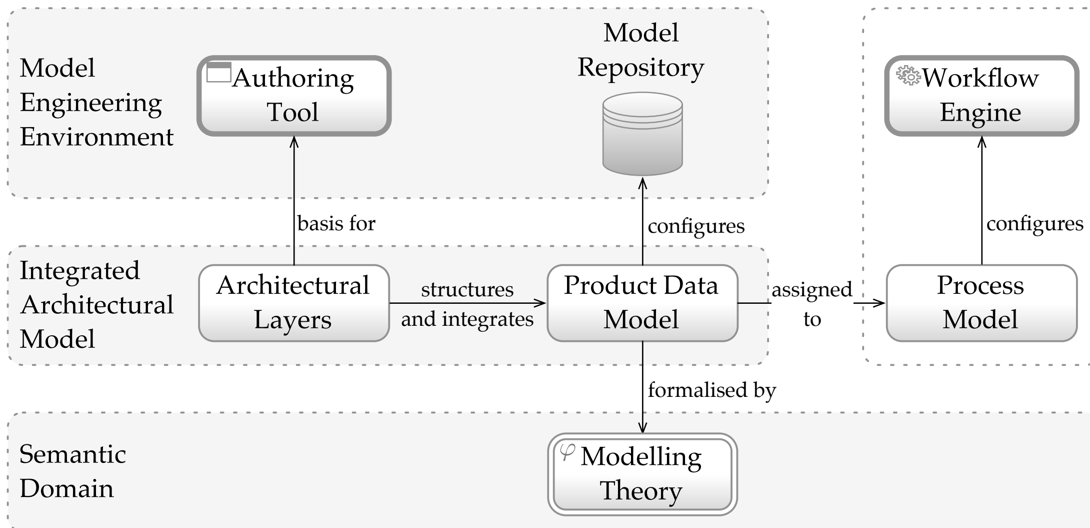



**Figure 3.4.:** Architecture of the COLA engineering environment as it has been realised (except the process and workflow part). According to Haberl et al. [86, 87] and Broy et al. [33].

aggregate necessary data from different data sources.

(iii) As all information are stored in a single repository, model analysis techniques, as for instance model checking, static analysis, or theorem proving, can easily be performed in a batch processing mode twenty-four-seven. By decoupling data storage from the modelling client, these time-consuming tasks can be run without affecting the client's performance. The proposed and realised tool architecture is depicted in Figure 3.4.

(iv) The way of collaboration between OEM and Tier 1 suppliers will change. Suppliers obtain via an access control mechanism the possibility to directly access relevant information and also to directly create, change, and delete artefacts. Of course, the allowed actions can be configured and also may change depending in the current process step. The same holds for the views, i. e., the artefacts that can be seen and also their representation. This tight coupling of OEM and suppliers simplifies and in turn accelerates the development process.

(v) The presented centralised approach enables the installation of a well-defined concurrency control, which is indispensable whenever hundreds or even thousands of engineers are working concurrently from inside or outside the company on the same models. Together with a process engine, a systematic

distributed engineering will be facilitated. All process activities carried out by different stakeholders are coordinated and orchestrated according to the current process step.

## 3.5. Summary

This chapter discussed the idea of seamless model-driven development of software-intensive systems. Through modularisation and hierarchisation of models and modelling of those along different levels of abstraction, a reduction of complexity apparent to engineers and a separation of concerns is facilitated. Due to the well-defined mathematical foundation of the COLA modelling language a fertile soil for the use of formal methods like model checking is created. Seamless modelling does not restrict itself to only modelling of artefacts, but also to store them in a repository implementing a unique product data model. Hence, there is no need to write adapters etc. to bridge the integration gap arising when using different tools with—and that is the crucial point—semantical different product data models.

CHAPTER

# FOUR

# The COLA Automotive Approach

This chapter summarises the COLA development approach for software-intensive automotive systems. The fundamental ideas outlined in Chapter 3 are instantiated in an effective implementation—the COLA-IDE. After an introduction given in Section 4.1, a concrete instantiation of the abstraction hierarchy is given in Section 4.2. The formal foundation will be COLA, which for its behavioural modelling is based on a synchronous data-flow language. The different levels of abstraction of COLA's architectural model are bridged in the development process explained in Section 4.4. Finally, Section 4.5 concludes this chapter with a discussion of related work.

**Contents**


4.1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

4.2. Architectural Levels . . . . . . . . . . . . . . . . . . . . . . . . . 56

4.3. COLA—The Component Language . . . . . . . . . . . . . . . . 66

4.4. Deployment Process . . . . . . . . . . . . . . . . . . . . . . . . . . 77

4.5. Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80


## 4.1. Introduction

The COLA automotive approach has been realised in cooperation with different departments of the BMW Group, including Research and Technology and E/E Processes, Methods, and Tools. Besides the informatics department with colleagues from Software & Systems Engineering, Theoretical Computer Science,

and Operating Systems, also the chair of Integrated Systems from the department for electrical engineering and information technology of the Technische Universität München was involved. During the collaboration of more than four years, a number of case studies have been developed that substantiate the practicability and viability of the COLA automotive approach. Details on the case studies are given in the Chapters 8.1 to 8.3.

This chapter describes the interplay of a well-suited architectural model, which facilitates modelling of artefacts on different levels of abstraction with a semantically and syntactically well-defined modelling language—The COmponent LAnguage, COLA . Furthermore, a highly sophisticated authoring tool coupled with a central repository not only enables efficient development, but also provides analysis capabilities to guarantee model consistency. As COLA's core is mathematically sound, formal methods like model checking, test case generation, and simulation, as well as fully automatic deployment are enabled.

**COLA—The Component Language**

## 4.2. Architectural Levels

This section explains in detail the architectural model of the COLA automotive approach. Basically, the architectural levels are structured following the work of Pretschner et al. [162] and Broy et al. [32]. Within the COLA automotive approach and the COLA-IDE [87], modelling is performed along three architectural levels, namely

(i) *Feature Architecture*,

(ii) *Logical Architecture*, and

(iii) *Technical Architecture*.

In this lineup, the Feature Architecture is the most abstract, and the Technical Architecture the most concrete architectural level, i. e., from the first to the last mentioned, the completeness increases with regard to the implementation.

Next, the different architectural levels are detailed on and the question 'Which artefacts are modelled and how is modelling performed?' will be answered.

### 4.2.1. Feature Architecture

**Feature Architecture**

The most abstract architectural level is the so-called *Feature Architecture*. Its principles are found in Rittmann [170], which in turn follows the ideas of FODA

(**F**eature-**O**riented **D**omain **A**nalysis) by Kang et al. [112]. The Feature Architecture as the starting point in the COLA automotive development approach is considered with requirements engineering and deals in the sense of Zave with the functions or features of a system under development and their relationship.

> Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families." [202]

Systems have become so complex during the last decades that a purely text-based requirements engineering and management turned out to be infeasible. The limits of what can reasonably be done with a text processing tool have been exceeded, even for—how Weber and Weisbrod [196] call them—*local heroes*.

Success of a software project comes along with a well-realised requirements engineering phase. According to Lutz [141], about 75% of failures found in operational NASA software are rooted in requirements errors. As the aeronautic domain poses similar demands concerning product quality, one can assume that these numbers also hold in the automotive domain, where traditional requirements engineering is done less systematic. Thus, the importance of requirements engineering for a successful project that finishes within the cost and time bounds, has to be emphasised. Wrong decisions made in early phases are particularly problematic in the automotive domain. As functions or features, which are synonymously used within this thesis, are oftentimes realised in a tight connection of a hardware *and* a software part, late changes are difficult to accomplish and in any case very cost-intensive. The tendency of the mentioned failure rate is confirmed by MacKenzie's [143] statement that source code 'is seldom the weakest link in the chain of dependability'. This analysis showed that coding errors contributed with only 3% to software errors—other than one might expect.

As the name of the level of abstraction at hand suggests, so-called *features* and their relationship are modelled on this level of abstraction. Unfortunately, there is no uniform definition of a feature in the literature. Therefore, the following definition is used for the remainder of this thesis.

**Feature**

**Definition 1** (Feature)**.** *A* feature *encapsulates a functional part of the system and is characterised by an interaction between a customer and itself capable of being experienced.*

Within this definition, the customer is not necessarily a driver of a car. In fact, also service personnel or even diagnosis devices may be customers of a feature.

Besides the features themselves, their relationship and interaction is modelled, too. This leads us to the following definition according to Zave [203].

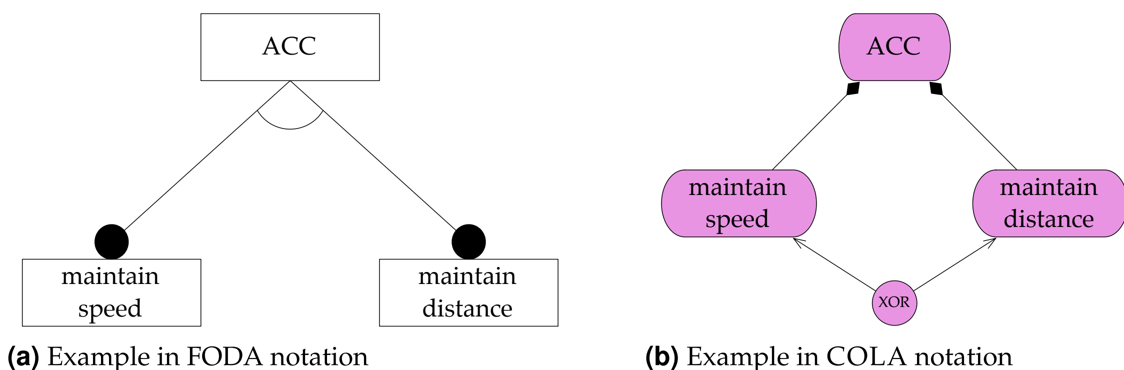
**Feature interaction**

**Definition 2** (Feature interaction)**.** *A* feature interaction *is some way in which a feature or features modify or influence another feature in defining the overall system behaviour.*

The aim of the Feature Architecture is to bridge the gap between informally written requirements and a formal, functional model. The Feature Architecture, thus can be seen as a black box only specifying the functionality visible at the system's interface. A common source of errors are unconsidered interactions between features. To avoid such errors, the designer can explicitly define these interactions and hence lift them on a formal foundation for later analysis. As the Feature Architecture has a functional rather than a technical emphasis, it does not consider and abstracts from any technical details like distribution, operating system, or hardware architecture. The specification on the Feature Architecture may also be partial and behavioural specifications may be non-deterministic.

**Feature Hierarchy**

**Structure and Structuring.** The first step of constructing the Feature Architecture is to build a so-called *Feature Hierarchy*. A Feature Hierarchy is a hierarchical decomposition of features into sub-features, yielding a complexity reduction amongst systems with rich customer functionality. A decomposed feature is also called a *combined* feature if and only if it is decomposed into at least two sub-features. Features that are not further decomposed are called *atomic* features. An example for feature decomposition is given in Figure 4.1. In this simple example,




**(a)** Example in FODA notation **(b)** Example in COLA notation

**Figure 4.1.:** Figure (a) depicts the feature diagram in the classical FODA notation. In contrast, Figure (b) uses the COLA notation making interactions explicit.

assume we have an adaptive cruise control (ACC) feature that is decomposed

into the sub-features 'maintain speed' and 'maintain distance', both sub-features are exclusive, i. e., only one can be active at the same time. Figure 4.1a depicts the feature diagram in the classical restricted FODA notation. In Figure 4.1b, the COLA notation is shown. In contrast to the FODA notation, COLA allows to mathematically model arbitrary computable relations, not only OR or XOR, as shown in the example. For instance, features can enable, disable, or interrupt others. All its sub-features together with the feature interaction determine the behaviour of the 'ACC' feature.

In the hierarchical structure of the Feature Hierarchy, which except for feature interactions is a directed tree, the root node represents the complete system under development decomposed into its sub-features.

Points to start constructing the Feature Architecture are different kinds of requirements. Usually, there are hundreds of more or less structured requirements specification documents, collections of scenarios, use cases, etc. The following two activities are involved:

1. Specification of the Feature Hierarchy

2. Specification of the Feature Architecture

In the first step, the user behaviour is decomposed in a top-down approach into sub-features, which again can be decomposed. The decomposition can be function-driven or mode-driven. A function may have a 'desired', an 'undesired', and a 'diagnosis' behaviour, which could result in three sub-features. Moreover, the Feature Hierarchy can be organised according to operating modes. In the example given in Figure 4.1, there are two modes 'maintain speed' and 'maintain distance'. The second step in the first activity is to model the feature interactions and finally to describe them with their functional requirements.

Based on the Feature Hierarchy, the formal Feature Architecture is modelled in the second activity. This includes the definition of the features' syntactical interfaces as a subset of the total system's interface. Their behaviours are specified within the semantical interfaces. This is done using two ways: first, the behaviour can be modelled using the COLA core modelling formalisms, i. e., data-flow networks and automaton (cf. Sections 4.3), or second, using SALT (**S**tructured **A**ssertion **L**anguage for **T**emporal Logic) [19] formulae to reason also over temporal properties. As SALT is not a major topic of this thesis, only those parts necessary for a better understanding will be explained at the relevant place. The first-mentioned is also referred to as a *constructive* approach, whereas the latter is called *descriptive* approach. The behavioural specification is done in a bottom-up

fashion, i. e., beginning with the atomic (leaf) features, combined features are constructed based on the sub-features until the root is reached. Then, the root contains the (partial) behavioural specification of the complete system. The constructive parts of the root, i. e., complete system feature, can already be used to generate test cases or to simulate the possibly partial behaviour. After the Feature Architecture has been finished, for those parts specified using COLA, a simulatable model is at hand. As the Feature Architecture is the closest architectural level with respect to the requirements one can use in this setting the notion of *executable requirements* according to Zave and Yeh [204].

### 4.2.2. Logical Architecture

If the Feature Architecture has been defined using COLA networks and automata, its root feature can be used as a first version of a Logical Architecture. However, as mentioned above, the Feature Architecture can be partial and possibly non-deterministic. Hence, it is necessary to resolve possible issues on the *Logical Architecture*, i. e., if the non-determinism was not intentional. As the model of this level of abstraction will be used to generate source code, it has to be complete and deterministic. Predictability of the modelled systems is very important. Of course, code can be generated from non-deterministic models and also simulation can be performed. However, in these cases, the language semantics has to have rules how to interpret non-deterministic models. In MATLAB/Stateflow, for instance, the so-called 12 o'clock semantics is used. There, transitions are taken with respect to their graphical layout in a clockwise direction, which cannot be seen as a good practice to prevent engineers from making modelling mistakes.

**Logical Architecture**

In contrast, to the Feature Architecture, the Logical Architecture is not structured with respect to functionalities, but in terms of architectural design. Here, also aspects like the organisational structure, dependability, maintainability, and reusability play a crucial role. Concerning the first mentioned, Conway [49] pointed out in 1968 that

**Conway's law**

> “[...] organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations”.

Thus, when taking the modelled root feature from the Feature Architecture over to the Logical Architecture, usually it has to be restructured to fulfil the mentioned criteria. Hence, on the Logical Architecture the following activities are involved:

(i) (re-)structuring,

(ii) completion of the behavioural model,

(iii) introduction of data sources and sinks, and

(iv) determinisation of undesired non-deterministic automata.

In addition to this top-down procedure, the Logical Architecture can also be designed 'from scratch'. If already existing sub-systems are reused, modelling follows a bottom-up approach.

Models on the Logical Architecture are constructed using COLA's core language constructs: units, data-flow networks, and automata (cf. Section 4.3). It should not be suppressed the fact that knowing that a model is complete is a difficult task. Zowghi and Gervasi [206] mention that

> "Davis states that *completeness* is the most difficult of the specification attributes to define and *incompleteness* of specification is the most difficult violation to detect [52]".

As the Feature Architecture speaks about usage behaviour, i. e., the observable system interface, additional logical sources and sinks have to be added in order to establish the connection to the physical world. Sources and sinks are the logical representation for sensors and actuators on the Technical Architecture, which will be discussed later. As it cannot be guaranteed that signals apparent at the system boundary on level of the Feature Architecture are also isomorphic present on the Logical Architecture, source post-processing and sink pre-processing functions may be needed. Assume for a moment the signal 'speed' on the Feature Architecture. There might be different technical realisations to obtain the speed of a car, for example, using GPS or calculating the speed based on the wheel speed and radius. Additional computations are needed accordingly.

Again, modelling of hierarchies using networks that contain sub-networks or automata whose states again contain networks, is supported. Hence, separation of concerns is facilitated through modularisation and hierarchisation. An example is given in Figure 4.2. In contrast to the most abstract level, the Feature Architecture, the Logical Architecture is complete in terms of the desired behaviour. Models on that level of abstraction are useful to apply formal methods such as for instance model checking (cf. Section 3.3).

### 4.2.3. Technical Architecture

**Technical Architecture**

The purpose of the *Technical Architecture* is to provide an abstract model of the target execution platform, on which the COLA model will be executed. The

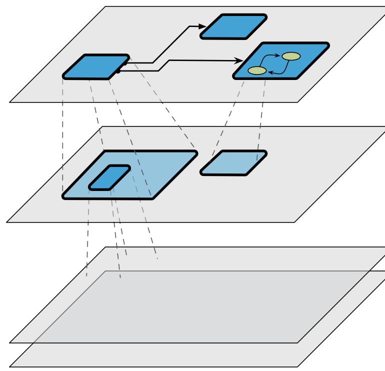


**Figure 4.2.:** Hierarchical decomposition of COLA models.

Technical Architecture is the most concrete level within the abstraction hierarchy of the COLA automotive approach. The aim is to provide as much information about the hardware topology and the run-time configuration to be able to deploy COLA models. For this purpose, the Technical Architecture consists of the three concepts

(i) *Cluster Architecture*,

(ii) *Allocation*, and

(iii) *Hardware Topology*.

Their interplay is visualised in Figure 4.3 and details are given next.

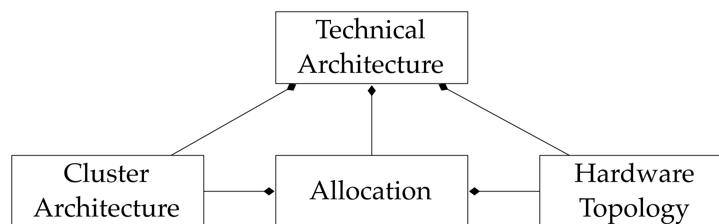



**Figure 4.3.:** Conceptual class diagram of COLA's Hardware Architecture.

### Cluster Architecture

Cluster Architecture

The transition form the Logical Architecture towards the Technical Architecture passes the so-called *Cluster Architecture*. The Cluster Architecture can be seen as an independent level of abstraction, or as a part of the Technical Architecture as it is realised here.

The COLA automotive approach deals with networked, distributed automotive systems. The Logical Architecture, however, defines only a single model. Thus, in order to execute the modelled behaviour distributed over a couple of ECUs, the model has to be cut into atomic pieces (tasks from an operating system's point of view). These pieces are referred to as *clusters* and define in their interplay the system's run-time behaviour. As clusters are atomic, they form the deployable entities that are considered during allocation and scheduling explained in the Sections 7.2 and 7.3.

**Cluster**

From a graph-theoretic perspective, the model on the Logical Architecture can be considered as a directed graph consisting of nodes—the COLA units—and edges—the data-flow channels. If the model is partitioned into clusters, data-flow amongst clusters is defined by the cut between the contained units. Figure 4.4 illustrate the circumstance. Note, only data-flow between clusters is depicted. There are three ways to build a Cluster Architecture:

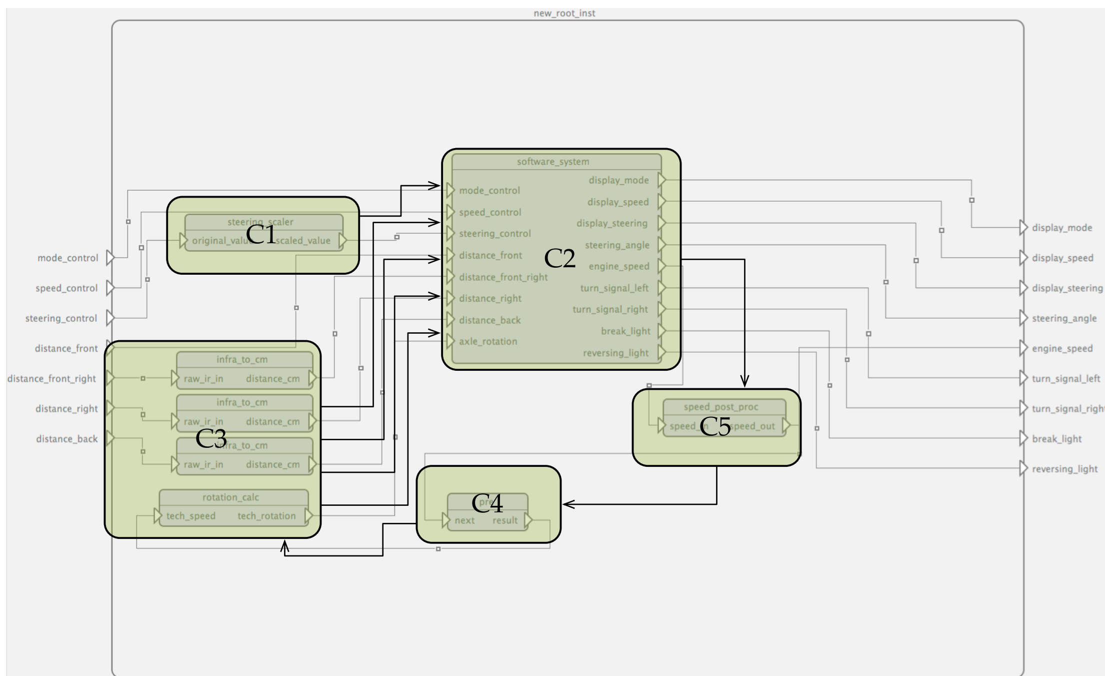



**Figure 4.4.:** Example of a possible clustering of the Logical Architecture into five clusters C1 to C5.

1. Engineers define manually the clusters and—if desired—also their placement on components of the Technical Architecture. This way is preferred if the emphasis is on a hand-crafted software architecture.

2. Engineers can use a mechanism to automatically generate clusters based on a heuristic, which tries to balance the cluster sizes with respect to the number of contained units.

3. Furthermore, a combined approach is possible. This means, a partial Cluster Architecture can be predefined by an architect, and is completed using automatic methods.

As clusters are atomic entities that are executed on the target hardware platform, they are the starting point for code generation and resource estimation. Whereas the Logical Architecture models the system behaviour, the Cluster Architecture defines the software architecture of the system under development and thus cuts the modelling artefacts from the Logical Architecture into deployable entities on the Cluster Architecture.

As soon as clusters are defined, executable code is generated for each of them. Note, the interplay amongst them is automatically given by the data- and control-flow dependencies of the logical model. The generated C code serves multiple purposes:

(i) It is used to estimate the performance figures using SciSim by Wang et al. [194]. SciSim combines the instrumented C code and micro-architecture simulators to model run-time interactions between software and micro-architectures. SciSim is used for each cluster and each ECU with their respective instruction set architecture, like for instance PowerPC, ARM, or SPARC, yielding performance figures. These are used for allocation and scheduling decisions during deployment.

(ii) Of course, the generated C code will be productively used 1:1 on the target system.

### Hardware Topology

**Hardware Topology**

The *Hardware Topology* models sensors, actuators, ECUs, and buses and their interplay. Buses connect different ECUs that can also act as gateways. Sensors and actuators are assumed to be hard-wired to ECUs. The mentioned basic concepts of the Hardware Topology are modelled each with as much information as necessary for scheduling and allocation.

In Figure 4.5 the hardware topology of the autonomous parking system (cf. Chapter 8.2) is shown. It consists of three ECUs that are connected via a single bus. Each ECU has attached some sensors and actuators and a bus interface.

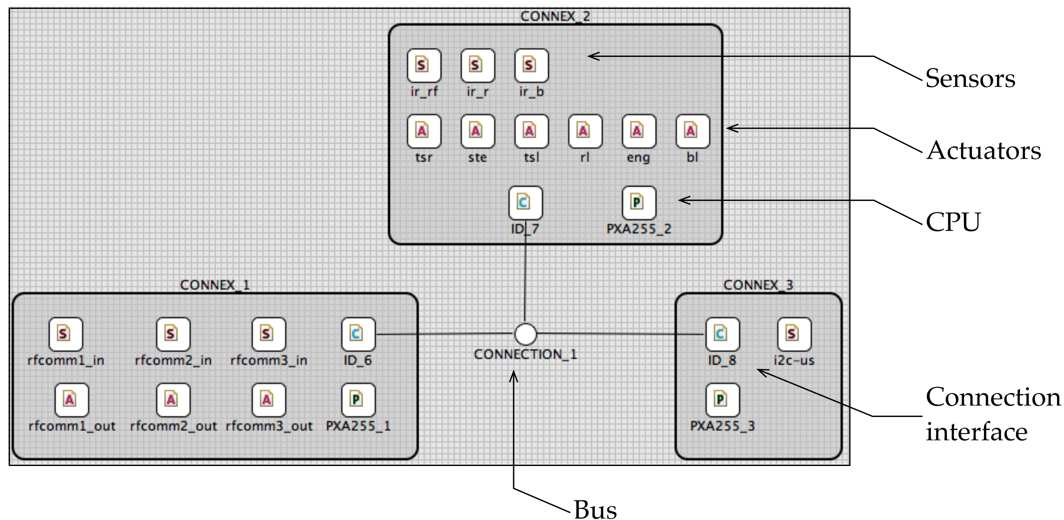



**Figure 4.5.:** Technical Architecture as modelled in the COLA-IDE.

### Allocation

The Cluster Architecture models *what* to deploy whereas the Hardware Topology defines *where* to deploy. The missing link between both is the *Allocation*. This concept specifies which artefact of the Cluster Architecture is deployed onto which artefact of the Hardware Topology. Clusters are mostly allocated onto ECUs, but also sensors and actuators are valid allocation targets. This is interesting, if complex intelligent sensors and actuators are modelled. The detailed allocation algorithm with its optimisation scheme is given in Section 7.2.

## 4.2.4. Summary

Figure 4.6 depicts the concrete COLA automotive approach instantiation of the abstraction hierarchy. By structuring the features and describing their possible interactions, the Feature Hierarchy is build in a first step followed by a bottom-up integration of their behaviours. By linking the features onto the Logical Architecture, which is usually done in an m:n relation, i. e., a feature is realised by multiple logical components, which in turn encapsulate the scope of possibly many features. The components of the Logical Architecture, i. e., units, can be decomposed hierarchically to structure the system and hide complexity apparent to developers. The next step cuts the logical model into clusters, which are allocated to entities of the Hardware Topology on the Hardware Architecture. Herby performance attributes and hardware capabilities are considered. Modelling along these levels

facilitates the separation of concerns, as it allows modelling of different aspects relevant for the respective step during the development process. During the development of automotive systems, usually different experts are involved. This includes, requirements engineers, function specialists, architects, and hardware component experts, just to mention a few important ones. Not everyone needs to have all information of and access to the model. Instead, only a special view or level is needed, which is also supported using the presented levels of abstraction.

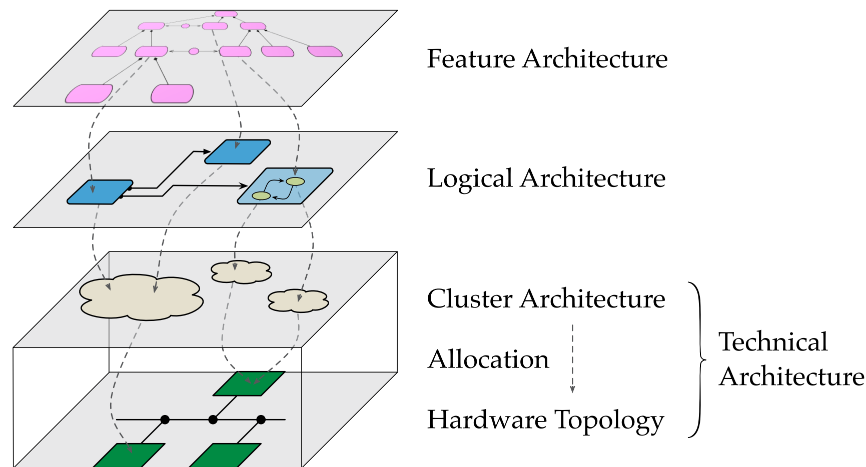



**Figure 4.6.:** Different levels of abstraction in the COLA automotive approach.

As clusters can be considered as tasks from an operating system's point of view, the separation of the Logical and the Cluster/Technical Architecture, solves two problems mentioned in Section 3.1:

1. Problem **P1** is solved as we have a separation between the functional and the architectural model represented by the Cluster Architecture (which basically represents the software architecture of the developed system).

2. Problem **P2** is solved as the Cluster Architecture can be considered as a task model with its annotated resource requirements.

## 4.3. COLA—The Component Language

In the previous chapter, modelling along different levels of abstraction has been discussed. On each level of abstraction different aspects of a system are modelled using dedicated modelling constructs provided by the COLA modelling language. One can distinguish between the operational part of COLA and a descriptive

part. The operational part is used on the Feature and on the Logical Architecture. It is denoted operational since it is based on an operational semantics, which is defined in [130]. On the two mentioned levels of abstraction, it is also possible to simulate models. The constructive part of COLA is also referred to as COLA *core* within this thesis. The non-constructive part is used to describe the mapping from logical COLA models onto entities of the Hardware Architecture, done within the Cluster Architecture. Properties of the target hardware including the topology are defined on the Hardware Architecture, too. In the following section, the basic concepts of COLA core are explained.

### 4.3.1. Basic Concepts

Being a *synchronous* data-flow language, COLA follows the hypothesis of *perfect synchrony*. This claims that models are executed in zero-time, i. e., we assume that computation does not need time. Furthermore, communication between modelled components does not need time either. Besides the zero-time assumption, logical components (from now on referred to as *units*) are executed in principle in parallel. Of course, this is only true for models of non-communicating units, i. e., where no data exchange between units occurs. Generally, there are networks of interconnected units exchanging data via so-called *channels*. In addition to *data-flow dependencies*, COLA provides the powerful concept of automata in general and *mode automata* in particular, which also imply causal relations, namely *control flow dependencies*. Automata model finite state machines similar to Statecharts [92]. Details about mode automata will be given in Section 4.3.2 (cf. also [18, 146]). The combination of units and communication channels is used to build complex data-flow networks. Following the ideas to structure and modularise systems, COLA allows to hierarchically decompose systems: complex COLA systems are built using basic, primitive operators. These basic units are atomic and cannot be further decomposed. They define basic Boolean and arithmetic operations, which are building blocks for complex mixed arithmetic and Boolean functions. Each unit has a syntactical input and output interface defined by typed input and output ports. Compatibility is checked and types are inferred as described by Kühnel et al. [131]. Channels are used to connect compatibly typed ports to realise data-flow. To do so, a single source port can be connected to at least one compatible destination port. COLA's semantics makes interruptions within feedback loops compulsory. This is necessary to ensure strict causality. Hence, on each data-flow path starting and ending at the same unit, at least one so-called *delay block* has to occur. This unit is initialised with a predefined value. This

**perfect synchrony**

**COLA unit**

**COLA channel**

**COLA automaton**

**COLA network**

guarantees that COLA models are always well-defined in the sense that there always exist unique fixed-points for the recursive equation system induced by the data-flow (channels). Delay blocks also provide means for retaining data for a single time unit. This concept can be used to store data like in variables known from high-level programming languages. Moreover feedback loops are a commonly used and well-suited modelling construct in control systems. As delay units store data they are *stateful* blocks.

The evaluation of COLA core models follows a cyclic execution on a discrete time base, so-called *ticks*. From one tick to the next consecutive tick, the complete model is evaluated once, i. e., COLA systems produce and process results in accordance with this time base. Logical interaction with the underlying hardware is done using *source* and *sink* blocks, to read and write values, respectively. These special units are the logical representation of sensors and actuators modelled in the Hardware Architecture.

In the following, the basic modelling constructs (basic blocks, data-flow networks, and automata) are given and exemplified.

### Basic Blocks

The simplest units, so-called *basic blocks* are constants, i. e., they provide at each tick a constant, predefined value at their output port 'out'. Moreover, COLA supports the basic arithmetic operations $+$, $-$, $*$, and $/$. These binary operations provide at each clock tick at their output port 'result' the value

$$result = lop\ \textbf{operator}\ rop \tag{4.1}$$

where $lop$ is the left operand and $rop$ is the right operand.

As the delay block retards values by one clock tick, the values apparent at its output port 'result' are hence defined over the infinite sequence of discrete time steps $(s_j)_{j \in \mathbb{N}_0}$

$$result(s_j) := \begin{cases} initial & \text{if } j = 0 \\ next(s_{j-1}) & \text{if j > 0} \end{cases} \tag{4.2}$$

where $initial$ is the initial value of the delay block. The up to now mentioned basic COLA blocks are visualised in Figure 4.7.

Another set of units is that of *comparison operators* $<$, $\leq$, $=$, $\geq$, and $>$. To express inequality ($\neq$), *Boolean connectives* are used, i. e., $(x \neq y) \equiv \neg(x = y)$. Besides the negation also the other common operators $\wedge$ (AND) and $\vee$ (OR) are supported. The last mentioned two categories are depicted in Figure 4.8. The port type of

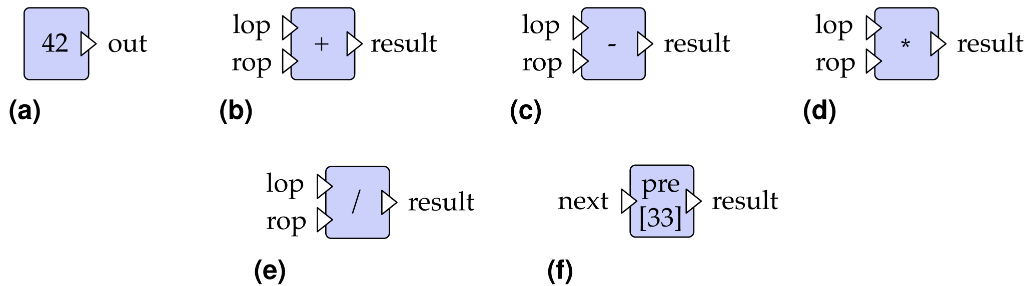



**Figure 4.7.:** Basic COLA blocks: (a) constant block with the value 42, (b) addition block, (c) subtraction block, (d) multiplication block, (e) division block, and (f) delay block initialised with the value 33, named 'pre'(vious).

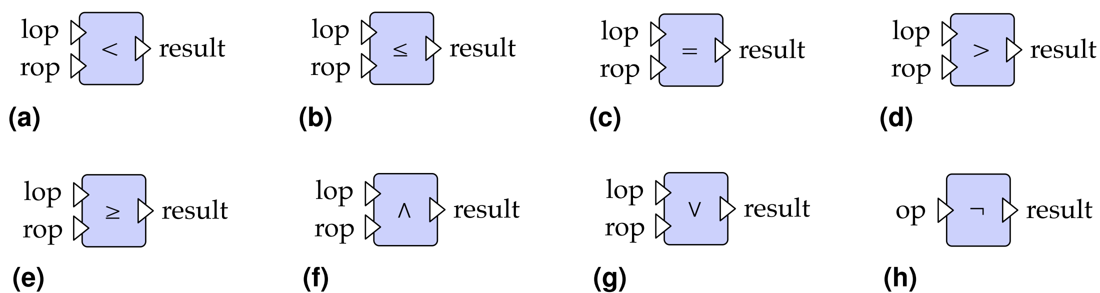



**Figure 4.8.:** Basic COLA blocks. (a) to (e) comparison operators and (f) to (h) Boolean operators.

Boolean operators is of course Boolean, which is also the type of the output port 'result' of all comparison operators. The input ports of all operators 'lop' and 'rop' are of compatible type as well as the corresponding 'result' port. The output port 'result' of a delay block has always the same type as its input port 'next'.

The last two basic blocks are *sources* and *sinks*. As the graphical representation of both a source and that of a sink block is similar to that of a constant block with an input instead of an output port, they are not depicted.

### Complex Data-flow Networks

Data-flow networks are employed to structure systems and thus reduce the complexity apparent to developers. They are used to hierarchically decompose systems from a high-level design down to functional details. The hierarchical decomposition facilitates a logical view on the system under development at different levels of abstraction. One of the simplest networks following the IPO-model (**I**nput-

**P**rocess-**O**utput) consists only of tree units 'Input', 'Process', and 'Output' as illustrated in Figure 4.9. The network 'IPO-model' thus consists of the mentioned

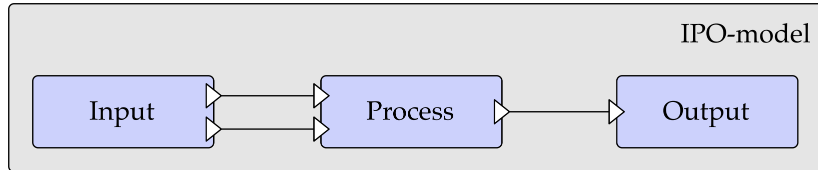



**Figure 4.9.:** High-level COLA design following the IPO-model.

units and three exemplary channels. The contained units are again networks. On the Logical Architecture level, channels are only allowed to connect a single source port with at least one destination port, thus realising a $1 : n$ connection. By descending networks, their originally hidden implementation becomes visible.

Networks are furthermore used to express the behaviour of automaton states and guards used as precondition for transitions. Automata are discussed in more detail in the following.

### Automata

COLA automata are special units that consist of states and transitions between them. They are basically finite state machines similar to Statecharts introduced by Harel [92] and also found in other modelling formalisms like the UML [26], Stateflow from The MathWorks, or ASCET-SD by ETAS Group.

Both, states and guards are themselves implemented by units: a state's behaviour is defined by a network, the guards are stateless networks, i. e., networks without occurrences of automata and delays since these units are *stateful*. They have to store their current state, in the case of an automaton, and their last value, in the case of a delay, for one execution cycle. As networks implementing the states's behaviour share the same signature, the current state, thus the network, is executed in place of the automaton. Moreover, networks implementing transition guards have the same input signature as the automaton, but, as they have to evaluate to **true** or **false**, they have a single output port of type Boolean. If and only if a transition's guard evaluates to **true** the transition is taken.

Starting from a dedicated state, the *initial state*, the semantics is defined as follows: let $q$ be the current state, if there is an outgoing transition whose guard evaluates to **true**, take it and execute the unit referenced by the target state. If there is no such transition predicate evaluating to **true**, execute the unit referenced by the current state $q$. Figure 4.10 gives an example of a data-flow **if**. Depending on

the input 'predicate' (Boolean value), either the value at the 'then' or that present at the 'else' port is written to the 'result' output port. The example also clarifies the circumstance of the unit interfaces. Moreover, it illustrates an algorithmic

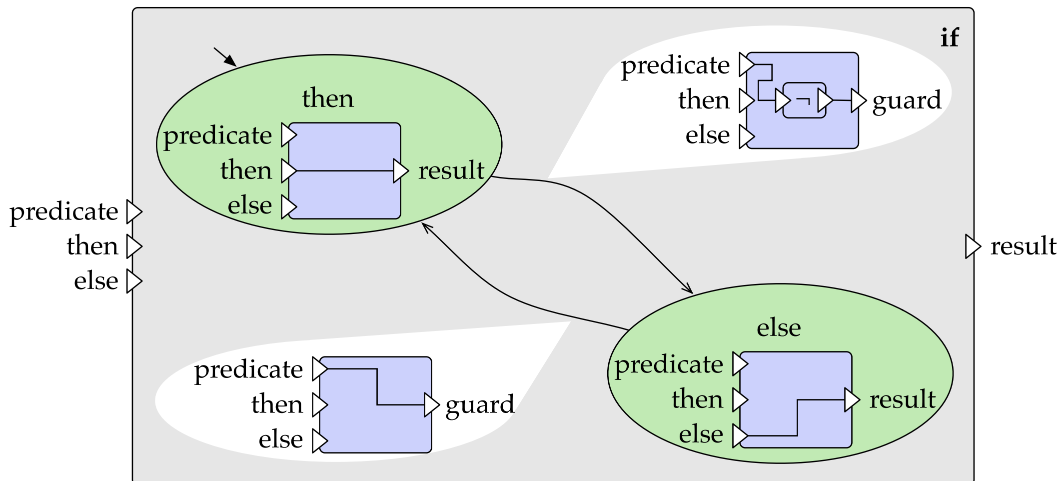



**Figure 4.10.:** COLA automaton modelling a data-flow **if**.

usage of automata. This is only one field of application. The second—at least as important—one is the application to decouple system behaviours in terms of *operating modes*, which will be discussed next.

## 4.3.2. Operating Modes

Using *operating modes* in embedded systems design in general and in software-intensive automotive E/E systems in particular has several advantages. But before stating them, the basic principle of operating modes is given. Maraninchi and Rémond propose to use mode automata to realise the notion of *running modes* for safety-critical systems development in [145,146]. The basic idea is to regard the system under development—be it a complete system like an aircraft or an automobile, or (parts of) functions—from a point of view that distinguishes between different modes of operation. Examples are for instance 'start-up', 'driving', 'accelerating', 'decelerate', or 'take-off' and 'landing', which Maraninchi and Rémond commonly give as example. With respect to electromobility or hybrid powertrains the use of operating modes will turn out to be a powerful modelling construct. Considering model-based engineering of embedded control software, Schätz [178] proposes a clear separation of control- and data-flow models to avoid unnecessary complexity. Mode automata are a way to model control-flow in complex COLA networks. Advantages are amongst others:

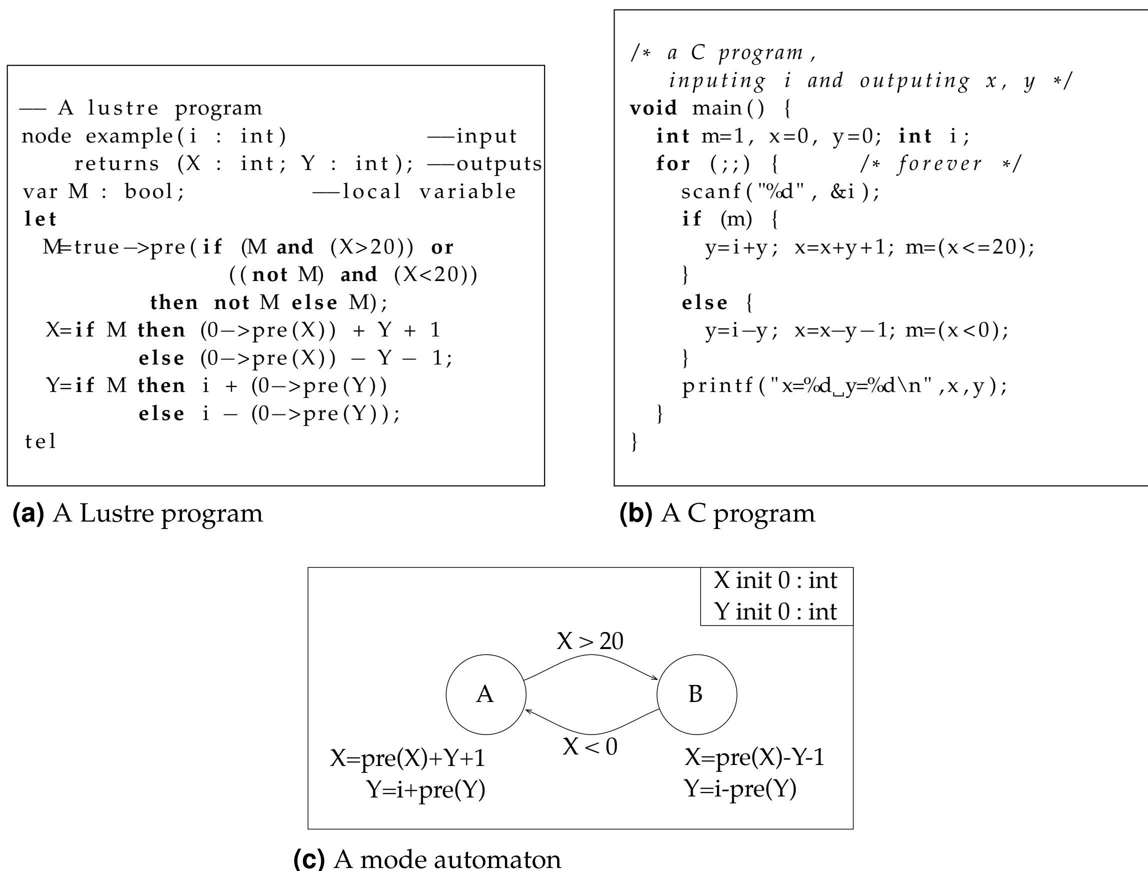
(i) separation of control- and data-flow,

(ii) complexity reduction by decoupling the behaviour (separation of concerns),

(iii) independent reasoning and analysis of modes,

(iv) facilitation of reuse, and

(v) reduction of system load since only a subset of all clusters is active, namely those corresponding to the current operating mode.




**(a)** A Lustre program

**(b)** A C program

**(c)** A mode automaton

**Figure 4.11.:** Figures (a) to (c) are taken from [146] and depict different implementations of a mode automaton: (a) Lustre code, (b) representation using C code, and (c) a graphical representation in automaton style.

The example given by Maraninchi and Rémond in [146] is depicted in Figure 4.11. The corresponding COLA implementation using mode automata follows the automaton of Figure 4.11c. Besides the textual description using Lustre, the

authors also provide C syntax. Depending on the current state, A or B, variables X and Y, both initialised with 0, are treated differently. The example uses a variable i, which in the COLA model (cf. Figure 4.12a) is realised using a constant block. Access to *pre*vious values in Lustre is modelled using delay blocks on the paths feeding back the values of variables X and Y, respectively. Accordingly, both are initialised with 0. Feedback-loops with delays make the use of previously computed values explicitly visible. Figure 4.12 depicts the corresponding COLA model. The figure shows the graphical representation of COLA. The textual representation of the automaton (cf. Figure 4.12b) is given in Figure 4.13. For a complete syntax and semantics specification, please refer to the appropriate COLA report [130]. The case study outlined in Chapter 8.2 uses the concept of

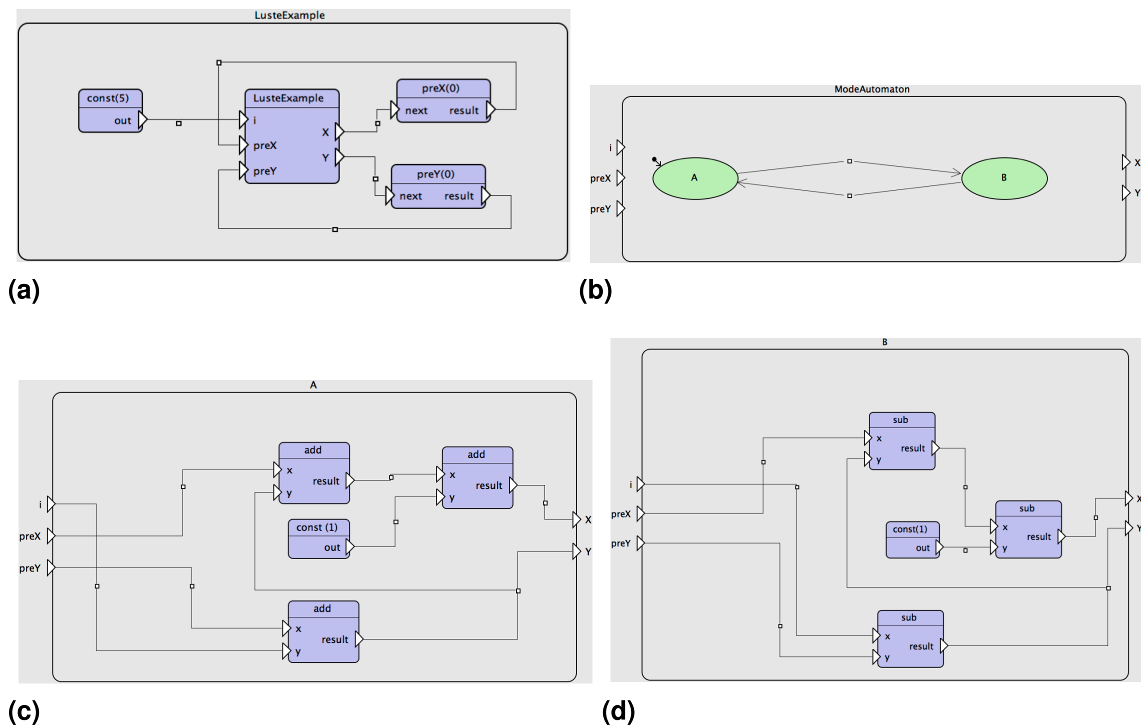



**Figure 4.12.:** Figures (a) to (d) show the hierarchical decomposition of the COLA model implementing the exemplary operating mode.

operating modes to decompose the system into the modes 'normal', 'parking', and 'sdc_active', i. e., a mode to manually steer the car, one mode that is responsible to find a parking space of sufficient size and to park automatically, and finally a mode with the behaviour: drive along a wall (with a constant distance) and elude obstacles.

```
automaton ModeAutomaton (i:Int, preX:Int, preY:Int -> X:Int, Y:Int) {
  initial state A {
    behavior network A (i:Int, preX:Int, preY:Int -> X:Int, Y:Int) {
      channel c6;
      channel c2;
      c2 := (preY+i);
      X := ((c6+c2)+1);
      Y := c2;
    }
  }

  state B {
    behavior network B (i:Int, preX:Int, preY:Int -> X:Int, Y:Int) {
      channel c1;
      c1 := (i-preY);
      X := ((preX-c1)-1);
      Y := c1;
    }
  }

  continue transition from A to B {
    guard network guard (i:Int, preX:Int, preY:Int -> guard:Bool) {
      guard := (preX>20);
    }
  }

  continue transition from B to A {
    guard network guard (i:Int, preX:Int, preY:Int -> guard:Bool) {
      guard := (preX<0);
    }
  }
}
```

State A with its implementing network

State B with its implementing network

Transition from state A to B

Transition from state B to A

**Figure 4.13.:** Textual COLA syntax of the automaton given in Figure 4.12b.

## 4.3.3. Syntax and Semantics of COLA

Below, some short notes about COLA's syntax and semantics are given. However, for more details please refer to the respective report by Kugele at al. [130].

### Syntax

COLA's graphical syntax is exemplified in numerous examples within this thesis and thus should be self-explanatory. In this context, Fuhrmann and von Hanxleden [77] point out the importance of a graphical syntax in model-based design. An example of the textual syntax is given in Figure 4.13. A complete definition in EBNF is given in [130]. One of the outstanding benefits of the COLA-IDE in

contrast to, for instance, MATLAB/Simulink, is that at each hierarchical level it is possible to edit the model using both the textual and the graphical model representation. In other tools—as the mentioned ones—it is *only* possible to use either the textual or the graphical representation. When defining the behavioural model on level of the Logical Architecture, it is beneficial in early phases to do the architectural design using the graphical syntax, whereas in hierarchically decomposed models, where at lower levels algorithmic details are of interest, it is usually more convenient for engineers to use the textual syntax.

### Semantics

COLA is based on a rigorous *operational* semantics. In [130], the semantics of COLA core has been defined by describing an interpreter for this synchronous data-flow language. This way was chosen to give the readers the opportunity to have a reference implementation for a COLA interpreter. This is useful for the development of tools building upon the COLA syntax and semantics. The following sums up the essentials of the COLA semantics: COLA models are cyclically evaluated based on a discrete time base with the assumption of perfect synchrony. Hence, all COLA units are executed once within a clock tick and in principle concurrently with respect to data-flow dependencies. Again, for further details please refer to Kugele et al. [130].

## 4.3.4. Examples from Control Theory

This section gives some examples from control theory showing the applicability of COLA. The first example is a Bessel filter, which is a linear filter heavily used in electronics and signal processing. Signal processing becomes more and more important in the context of automotive driver assistance systems. All new automotive features like for example the lane departure warning, the adaptive cruise control system, or the blind spot assistant have all in common that a lot of signals coming from sensors have to be processed. The second example is a PID (**P**roportional-**I**ntegral-**D**erivative) control which is very often used in control engineering

### Bessel-Filter

In Figure 4.14 a Bessel filter with the transfer function

$$H(z) = \frac{Y(z)}{X(z)} = \frac{4z^2 + 8z + 4}{12z^2 - 4z - 1}, \tag{4.3}$$

which is the quotient of the z-transformed input and output signals $X(z)$ and $Y(z)$, respectively, is given. The corresponding COLA model is given in Figure 4.15c. When stimulating the system with a discretised rectangular wave (cf. Figure 4.15a)

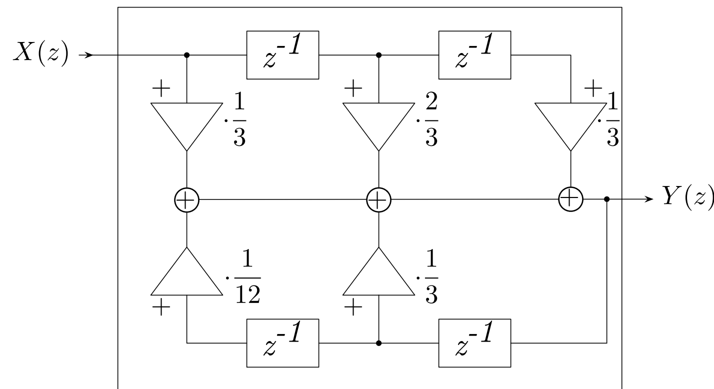


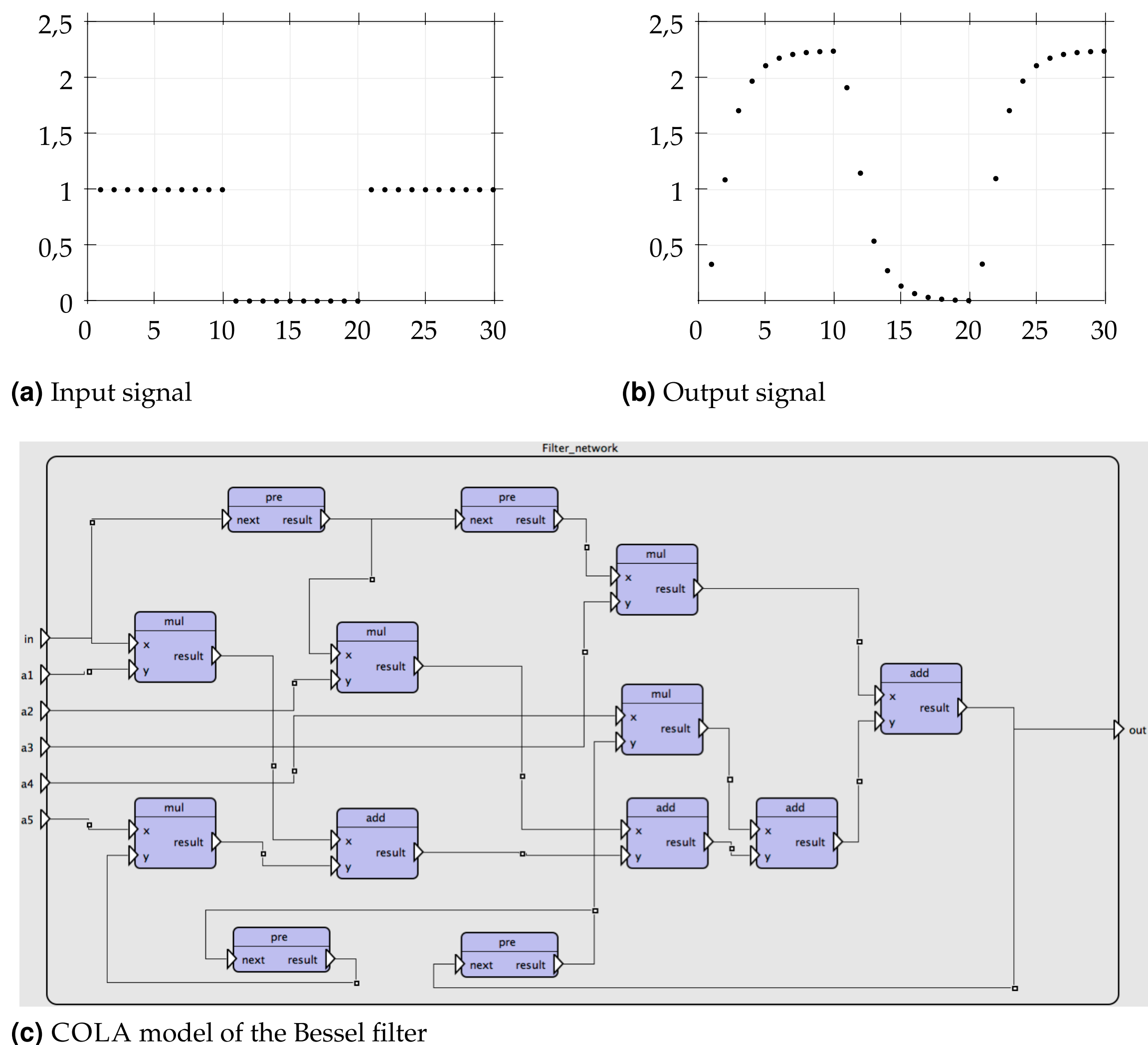

**Figure 4.14.:** Exemplary Bessel filter diagram.

the system responses with a signal depicted in Figure 4.15b. Note, the values are obtained using the COLA simulator. This example shows that COLA delay blocks are well-suited to model recursive filters, i. e., filters with feedback loops. The multiplication elements in the signal-flow graph are modelled as input ports so that their values can be adjusted, which makes it more generic.

### Proportional-Integral-Derivative (PID) controller

In cooperation with a project from the electrical engineering department, a test set-up has been developed (cf. Figure 4.16) that demonstrates the feasibility of the COLA approach. In this test set-up, a magnetic field is controlled in such a way that a metal insert floats at a specified point.

Therefore, an exemplary part of a larger MATLAB/Simulink model, namely a PID controller, was modelled using COLA. The PID block was realised as an S-Function whose C code was generated using the COLA C code generator. As the COLA model is complex and hierarchically structured, it can be found in the Appendix A (Figure A.1). Figure 4.17 shows the PID controller with a controlled system (general setting). $u(t)$ is the set value, $e(t) = u(t) - y(t)$ is the error (deviation between set and real value). $K_p$, $K_i$, and $K_d$ are proportional, integral, and derivative gain, respectively.




**(a)** Input signal

**(b)** Output signal

**(c)** COLA model of the Bessel filter

**Figure 4.15.:** (a) Discretised rectangular wave as input signal and (b) the corresponding output, i. e., $Y(z) = H(z) \cdot X(z)$. In (c) the used COLA model is depicted.

## 4.4. Deployment Process

One of the most fundamental ideas envisioned and then realised within the COLA automotive approach was *seamless model-based development* of software-intensive automotive systems. This development proposal contains several aspects. The modelling theory and development along different levels of abstractions have already been discussed. In order to bridge the gap from the Logical Architecture via the Cluster to the Technical Architecture some *process* steps have to be taken.

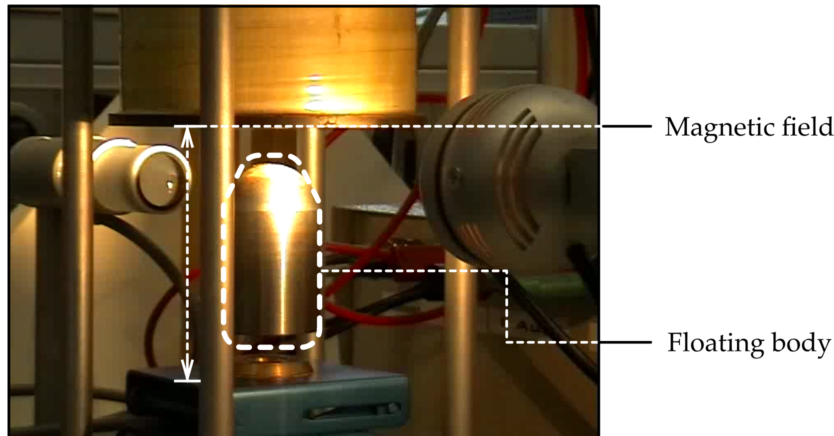



**Figure 4.16.:** Floating body in a magnetic field.

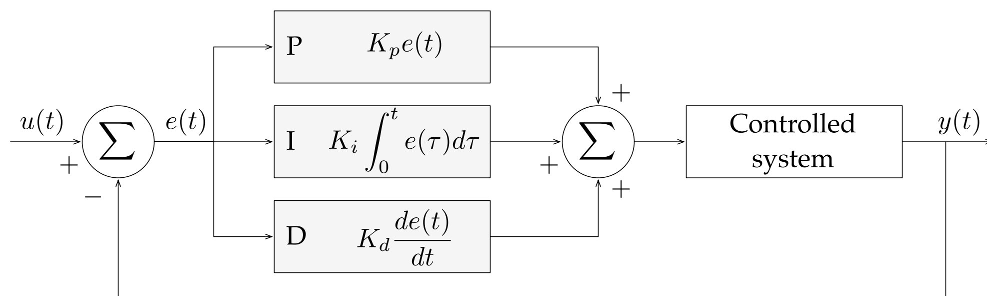



**Figure 4.17.:** PID controller with a controlled system.

In the following, the involved steps are shortly described and some are outlined in detail later on, namely *allocation* in Section 7.2 and *scheduling* in Section 7.3.

**(S0)** *Modelling and Verification*: As a requisite, the logical model and the technical model have to be modelled. It is assumed that verification and validation tasks like model checking, simulation, and testing have been applied in order to guarantee a high model quality.

**(S1)** *Partitioning*: Once, the logical model, i. e., the Logical Architecture, has been completed and verified, it has to be cut into deployable, atomic entities, which—from a technical point of view—are tasks running on an E/E component. In the COLA approach, these entities are called *clusters* on level of the Cluster Architecture.

**(S2)** *C Code Generation*: The C code generation step has two different purposes: first, it is used to estimate the expected performance (worst-case execution

time) on the modelled target system. Second, the source code is uses to be executed on the actual target system. Details can be found in [84, 90].

**(S3)** *Resource Estimation*: The generated source code is used as input for the performance estimation tool SciSim presented by Wang et al. [194].

**(S4)** *Allocation*: Using these resource figures and the capabilities of the modelled hardware, an allocation, i. e., a mapping of clusters onto the available E/E components, usually ECUs, is performed. This procedure also includes an optimisation with respect to non-functional requirements (cf. also [129] and Section 7.2).

**(S5)** *Scheduling*: The mapping information together with the resource figures and a data-flow analysis yield the basis for schedule plan computation. Details are explained in Section 7.3 and [88].

**(S6)** *Configuration*: As the presented approach uses a middleware for transparent communication and clock synchronisation, each ECU has to be configured at start-up. Information about locality of allocated tasks and their schedules are set. Detailed information about the developed middleware is given by Haberl et al. [85].

Note, even though allocation and scheduling are divided in two steps (**S4** and **S5**), a single step would have been possible as well. However, a separated consideration makes each of the steps more feasible. Moreover, different technological realisations can be examined. As a point of criticism, one can state that a combined approach would lead to even better results. However, the advantages outweigh the disadvantages.

The outlined steps cannot be performed in a stringent way in any case, but loops back to previous steps have to be done due to several reasons:

**S0⟶S0** If model verification or validation techniques are applied and find errors or inconsistencies with respect to the specification, the model has to be fixed.

**S1⟶S0** If there is no partitioning yielding both a feasible allocation and a schedule plan satisfying all timing requirements, the model has to be changed.

**S4⟶S1** If no feasible allocation can be found, the model partitioning has to be changed.

**S5⟶S4** If there is no feasible schedule for the distributed system, the allocation has to be changed.

These cases are illustrated in Figure 4.18.

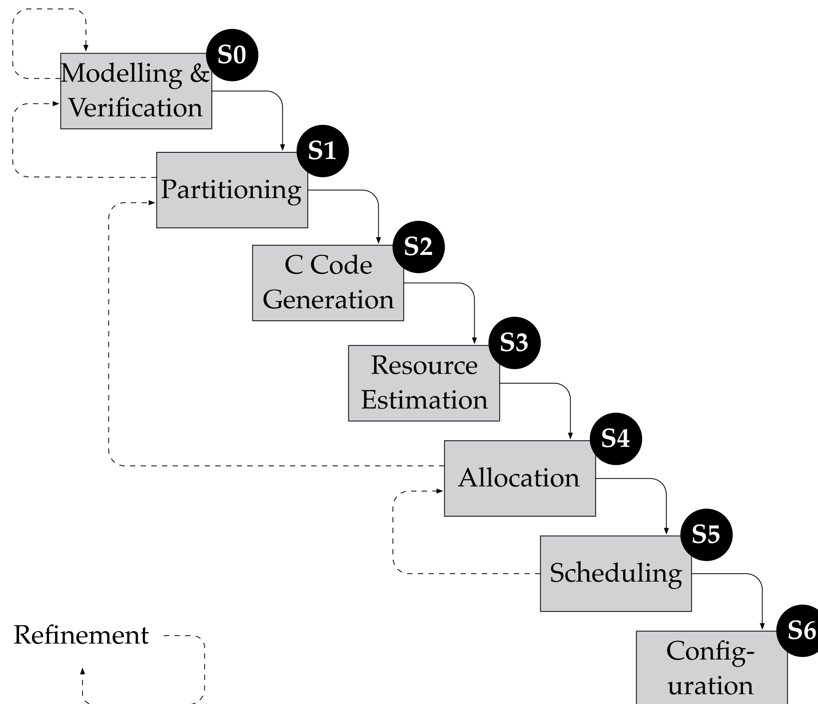



**Figure 4.18.:** COLA deployment steps.

## 4.5. Related Work

During the last years and decades, a lot of work has been done in the field of embedded systems development. Some of the approaches are to some extent similar to the COLA automotive approach. However, none of them is able to capture the development process from an early requirements specification to a deployment phase in a single seamlessly integrated tool. With respect to modelling along different levels of abstraction, the presented approach bases on work of different authors outlined below.

The following distinguishes between architectural design and behavioural modelling of software-intensive automotive systems, as the presented approach captures both while most of the available tools and methods do only capture either of them.

### 4.5.1. Modelling Along Different Levels of Abstraction

Modelling of software-intensive automotive systems along different levels of abstraction has been described by Pretschner et al. [162], Broy [30], and Broy et al. [32].

Wild et al. [197] propose to develop automotive software along four levels of abstraction, namely *Service Level*, *Functional Level*, *Logical Cluster Level*, and *Platform Level*. These levels build the fundament for an automotive specific architecture description language called *CAR-CL* (**C**ombined **AR**chitecture **D**escription **L**anguage). The presented approach in this thesis is quite similar to the just mentioned but in contrast, it considers the Cluster Architecture as part of the Technical Architecture. Similarly, the mobilSoft project [197] proposes a four-levelled approach. Van der Beeck [191] describes how to model a logical and a technical architecture for automotive systems using UML-RT. This approach does not consider the specification and structuring of customer requirements. EAST-ADL [65] (**E**lectronics **A**rchitecture and **S**oftware **T**echnology - **A**rchitecture **D**escription **L**anguage) is a modelling language first published in 2004. Current version is 2.1 and development is still in progress. It supports modelling of E/E automotive systems in four levels of abstraction, namely *vehicle level*, *analysis level*, *design*, and *implementation level*. Chen et al. [45] use EAST-ADL2 and show how to model safety-critical embedded systems. They describe how safety cases can be modelled and give a basic introduction into EAST-ADL2. Main drawback of architecture description languages in terms of a seamless and pervasive development is the missing capabilities to model behaviour. Other approaches to mention are the Save-IDE [183] (**Save** **I**ntegrated **D**evelopment **E**nvironment) and ModES [61] (**Mod**el-driven Design of **E**mbedded **S**ystems), both lacking behavioural modelling capabilities. Furthermore, there is also the Forsoft Automotive [27] project to mention.

**AutoFOCUS** **FOCUS**

AutoFOCUS [34, 100] is a CASE tool for the design and analysis of distributed, reactive, timed systems. As the name suggests, it is based upon the FOCUS [37] theory that provides a computational model based on the notion of streams and stream processing functions. Similarly to the presented approach, AutoFOCUS models are designed along the same three levels of abstraction only with a slightly different name albeit the same purpose: Functional Architecture, Logical Architecture, and the Technical Architecture.

**domain-specific language**

Considering UML, Broy et al. [35] state that UML does not base on a proper theory and favour to use a **D**omain-**S**pecific **L**anguage (DSL) to model both the structure and the behaviour of systems. One drawback closely linked to its insufficient semantics definition is the herefrom resulted inconsistent usage in the daily engineering practice. With the definition of the UML profile MARTE [155] (**M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded Systems) better support for embedded system development with its special requirements is given. For aspects concerning modelling of time in this context, refer to André and de Simone [54].

With regard to systems engineering, SysML [156] has to be mentioned. SysML as a graphical modelling language combines a subset of UML 2 with some extensions and is a response to the UML for Systems Engineering RFP developed by the OMG (**O**bject **M**anagement **G**roup) and INCOSE (**I**nternational **C**ouncil **O**n **S**ystems **E**ngineering). The **A**rchitecture **A**nalysis & **D**esign **L**anguage (AADL) [72] or Avionics Architecture Description Language as it was formerly known, is an architecture description language first developed in the field of avionics. It provides formalisms to describe both, software and hardware aspects and allow to analyse models in general and to perform schedulability analysis in particular using for example the OSATE [181] tool set.

Specifically tailored towards automotive software systems is AUTOSAR. This industrial partnership started in 2003 with the development of a standardised software infrastructure with the aim to control the increasing complexity of automotive systems. An extension of the presented approach towards AUTOSAR interoperability has been investigated by Haberl [84].

### 4.5.2. Behavioural Modelling

One of the main benefits of the COLA automotive approach is that, besides the definition of the software architecture—using the Cluster Architecture—and the specification of the Hardware Topology within the Technical Architecture it also supports behavioural modelling on the Feature and the Logical Architecture. Behavioural modelling is realised using the COLA core data-flow language with its two fundamental concepts:

(i) hierarchical decomposition in COLA networks (data-flow) and

(ii) control-flow specification using COLA automaton.

Both modelling concepts are, indeed, not new, but seamlessly and rigorously integrated into the language core. Established CASE tools like MATLAB/Simulink use data-flow networks for the description of complex automotive systems, and subsequent code-generation [186], just to mention industrial tools first. Conrad and Doerr point out in [48] the significance of a well-elaborated syntax and semantics definition and documentation. As a negative example, they mention the de-facto industry standard for modelling control systems MATLAB/Simulink/Stateflow. The manual of those tools describes syntax and semantics extensively, however it lacks rigour.

Predominant in the avionics domain is the commercial SCADE Suite [3], which is based on the synchronous data-flow language Lustre. The ASCET (**A**dvanced

**S**imulation and **C**ontrol **E**ngineering **T**ool) products [69] by ETAS Group are further tools widely used in the automotive domain for safety-critical systems like ABS, ESC, and the engine control unit.

Similar to the mentioned data-flow language Lustre, also COLA is based on the synchronous paradigm. Benveniste et al. [21] present a theory of synchronous data-flow languages. Many data-flow language have their roots in Kahn's [110] process networks.

Lucid Synchrone [161], was one of the first approaches to extend the synchronous paradigm towards a higher-order type system known from functional languages, and to express programs that would be expressive and at the same time executed synchronously. Another well-known synchronous approach, but instead of a functional character using an imperative character, is Esterel [24]. Esterel systems differ, in that the behaviour is defined in a reactive manner, rather than functionally based on the data-flow relations alone. Various efficient implementations of synchronous languages in the form of textual (e. g. Esterel, Lustre [43, 91], Signal [81, 134], and FOCUS [37]), or graphical languages as for instance AutoFOCUS [34, 100] exist. Actually, graphical modelling of AutoFOCUS is very similar to that of how COLA models are designed within the COLA-IDE. Moreover, from a semantics point of view, AutoFOCUS is based on the time-synchronous notation of streams, hence models designed herein are executed at a discrete time base (ticks) and logical components communicate via channels synchronously. In fact, both tools have inspired each other. When restricting MATLAB/Simulink to only discrete modelling blocks, it also adheres to the synchronous languages paradigm [171]. For an example of an efficient implementation scheme of synchronous data-flow programs cf. [172]. Moreover, with GALS (**G**lobally **A**synchronous, **L**ocally **S**ynchronous) [44] there exists another wide-spread approach to implement synchronous systems in a distributed and non-synchronous environment. GALS can be understood as having 'synchronous islands' sitting and communicating in an asynchronous environment.

CHAPTER

**FIVE**

# Model Analysis

This chapter highlights some techniques to lift the overall model quality to a higher level. After a brief introduction in the following section, Section 5.2 discusses a way to use model checking techniques to reason about requirements in an early development stage. Next, in Section 5.3 an SMT-based approach is presented to look for possibly undesired non-deterministic system behaviour, and finally in Section 5.4 a translation scheme is described to transform COLA behavioural models into models within the Coloured Petri nets notation. This allows us to use the full power of, for example, the CPN-Tools [50] to analyse COLA models.

**Contents**


5.1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

5.2. Requirements Analysis . . . . . . . . . . . . . . . . . . . . . . . 86

5.3. Deterministic Models . . . . . . . . . . . . . . . . . . . . . . . . 92

5.4. COLA Model Analysis via a Translation to Coloured Petri Nets 101


## 5.1. Introduction

Probably the most crucial benefit of formal models, besides synthesis and model transformation, is the ability to reason about models. The use of formal models or a formal description notation in general opens the door to the powerful tools used in the formal methods community. The toolbox contains amongst others static analysis, model checking, theorem proving, testing, and simulation. D'Silva et al. [62] give good overviews of automated techniques for formal software verification.

Depending on the current step during the overall development process, different techniques can be applied. Thus it is up to the engineer to select the appropriate verification technique(s). This is what Kordon et al. [122] mean by emphasising the *verification* aspect and relax the 'model centring'. They propose to move from model-driven engineering to *verification-driven engineering (VDE)* following a VDE helicoidal life cycle (cf. Figure 5.1).

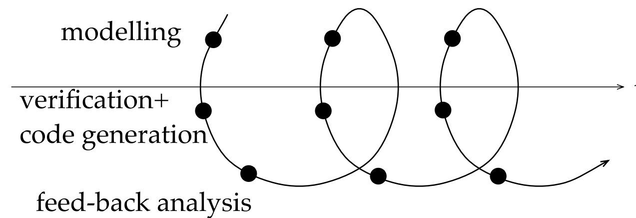



**Figure 5.1.:** The VDE helicoidal life cycle (according to [122]).

In this sense, the following Sections exemplify how verification techniques can be applied at different stages of development.

## 5.2. Requirements Analysis

During the development process of software systems in general and software for embedded, safety-critical systems in particular, the phase of requirements engineering plays a crucial role. Requirements engineering has to be performed very carefully with due respect. This is important in particular with respect to two exemplary surveys [70, 185], which identified poor requirements specification and management to be the reason for at least half of the problems during the development process. They include never completed projects, incomplete functionalities, major cost overruns, and significant delays. The expression of requirements in natural language is in many cases not sufficient especially when developing safety-critical system. Unfortunately, it is a common practice of requirements engineers to specify requirements not in a formal and precise manner. This may cause in inconsistent specifications, which is not acceptable. Therefore, rigorous requirements specification is done in the presented development process right from the beginning. However, there is still the possibility that formally specified requirements are inconsistent. Due to the formalisation, however, the full mathematical power of formal methods and techniques is available to limit the adversity and support the requirements engineer.

First, the problem statement will be introduced more formally in Section 5.2.1

before explaining in more detail the technical realisation in Section 5.2.2. Finally, a discussion in Section 5.2.3 concludes this chapter.

### 5.2.1. Introduction

Section 4.2.1 introduced the essential information about the Feature Architecture. As pointed out, the Feature Architecture is basically used to organise the informal given requirements usually given as text documents (requirements specification documents) by structuring the system from a feature or functional point of view. Together with so-called feature interactions, the relationship between different features is specified. This is an integral additional benefit, since informal requirements documents tend to be confusing, ambiguous, and redundant. Of course, the quality of such documents fluctuates with the experience of the requirements engineer. However, to come to his or her defence, it turns out that text documents are good for contractual purposes but not to provide a convenient and less error prone formalism, especially when considering the sheer length of such documents. An approach to automatically generate requirements specification documents in a model-based fashion is discussed in Chapter 6. The in this thesis presented approach proposes to specify features either in a

(i) *constructive* way, i. e., using COLA data-flow networks and automata, or a

(ii) *descriptive* fashion, i. e., using SALT, the general-purpose language for creating concise temporal specifications.

The following describes how SALT—in the descriptive fashion—is used to automatically detect inconsistencies between formalised requirements and therefore features. Even though the Feature Architecture with its Feature Hierarchy helps the requirements engineer to organise the customer requirements in a tree-like way and thus supports a clear arrangement, it is still possible that specified requirements are contradicting. Besides a feature description in natural language, a formal specification of that particular feature can be given. The ***S**tructured **A**ssertion **L**anguage for **T**emporal Logic* (SALT) is used for that purpose and illustrated in the following example.

Suppose, the following requirements, inspired by the case study given in Section 8.2, are given in natural language:

**(R1)** The side distance control shall always be active.

**(R2)** Steering movement (left or right) shall lead to a (left or right) direction of motion.

| Identifier | Description |
| --- | --- |
| `sdc_active` | Side distance control is active |
| `driver_left` | Driver steers to the left |
| `driver_right` | Driver steers to the right |
| `driver_sdc` | Driver activates the side distance control |
| `car_left` | Car moves to the left |
| `car_right` | Car moves to the right |
| `search_active` | The car is in the mode to search a parking space |

**Table 5.1.:** Identifier used within the SALT formulae.

**(R3)** The driver shall be able to enable the side distance control.

**(R4)** It shall only be possible to steer in one direction (left or right).

**(R5)** During steering, the side distance control shall be inactive.

**(R6)** When searching for a parking lot, the side distance control shall be active.

Assume, the requirements are spread over hundreds of pages and not listed like here. Each of the requirements considered for its own makes sense. However, in their combination they are inconsistent, i. e., they are contradictory. **R1** requires the side distance control system to be enabled permanently. However, this is contrary to requirement **R5**, which stipulates that the side distance control shall be inactive, whenever the driver performs a steering motion. If the mentioned requirements are formally specified, for instance with SALT as done within the COLA automotive approach, one can formally check them. The six requirements being in the style of the case study are formally specified as follows. The meaning of the used identifiers is explained in Table 5.1.

**(F1)**
```
(always ("sdc_active"))
```

**(F2)**
```
(always ("driver_left"  implies (eventually "car_left"))) and
(always ("driver_right" implies (eventually "car_right")))
```

**(F3)**
```
(always ("driver_sdc"   implies (eventually "sdc_active")))
```

**(F4)**
```
(never  ("driver_left"  and "driver_right"))
```

**(F5)**
```
(never ("sdc_active") between
incl req ("driver_left" or "driver_right"),
excl req not ("driver_left" or "driver_right"))
```

**(F6)**
```
(always ("sdc_active") between
    incl req ("search_active"),
    excl req not ("search_active"))
```

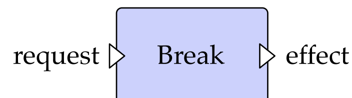
With this formalisation it is now possible to check whether they are inconsistent, where the formalised requirement **F***i* corresponds to the informal requirement **R***i*.

In the following, the process of detecting inconsistencies among a set of formally specified SALT requirements is discussed.

### 5.2.2. Realisation

SALT is a domain independent specification language, which, for instance, contrary to LTL (**L**inear **T**emporal **L**ogic) [160] is much more readable. In terms of expressiveness they are equivalent. In the following, the technical realisation of the analysis method within the COLA automotive approach is discussed. Note that this thesis assumes that features are attached with SALT specifications, i. e., their specification is given in the descriptive way.

Functional aspects of features are described in terms of their semantic interface. Thus, we talk about a subset of the complete system interface visible to customers—exactly that, which is necessary for service delivery. Thus, in principle only unique port identifiers are used as variables (identifiers) in SALT specifications. Consider the following simplified example for illustration. It should always be the case that after a breaking request (`request`) eventually a breaking effect (`effect`) follows.




```
assert always (request implies eventually effect)
```
(SALT)

```
G (request -> (F effect))
```
(LTL)

This example illustrated the connection between port identifiers and their usage within SALT specifications. Other than this example may convey, LTL formulae are usually longer and especially much more difficult to specify compared to their SALT counterpart.

The SALT compiler supports the translation from SALT specifications into several temporal logics depending on the specifications. When only temporal operators are used, LTL is the output format. However, if the so-called *timed layer* is used, TLTL is generated. In this thesis, however, only the first is considered. As the SALT compiler translates each SALT specification of a feature into a corresponding LTL formula $\phi$, consistency amongst them has to be checked. For

each LTL formula $\phi$ there is an effectively constructible Büchi automaton $\mathcal{B}$. We say that two features, specified by SALT ($\phi_1$ and $\phi_2$) and therefore LTL formulae ($\phi'_1$ and $\phi'_2$) are consistent, i. e., not contradicting, if the intersection of the respective Büchi automata ($\mathcal{B}_1$ and $\mathcal{B}_2$) recognises a non-empty language. That is, $\phi'_1$ and $\phi'_2$ are consistent if, and only if, $\mathcal{L}(\mathcal{B}_1 \cap \mathcal{B}_2) \neq \varnothing$. Figure 5.2a illustrates this circumstance for $n$ SALT specifications.

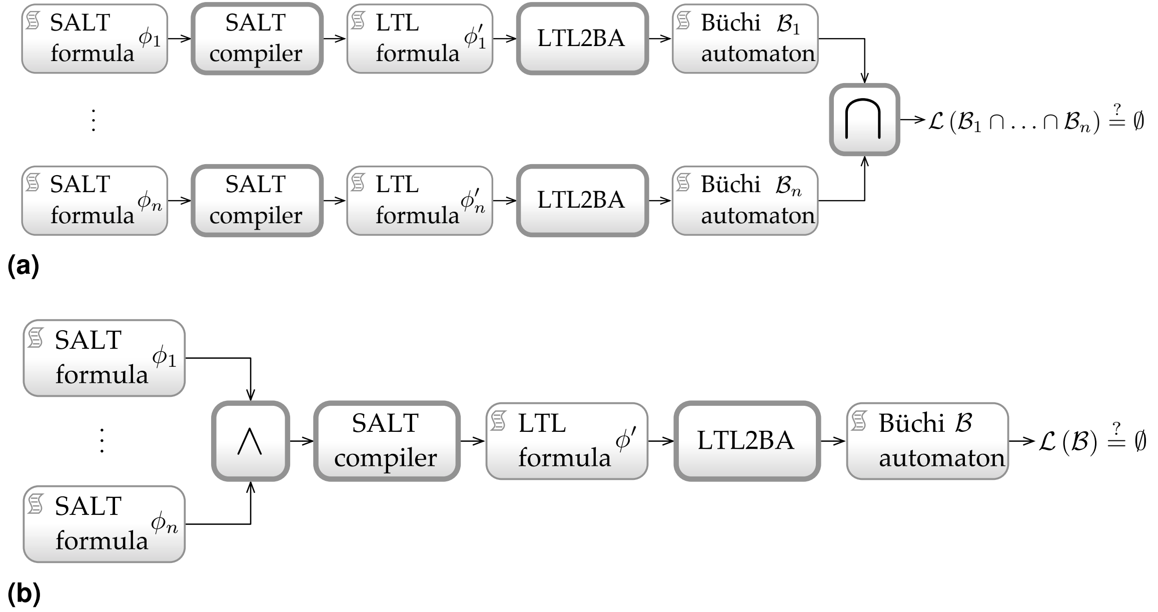



**Figure 5.2.:** (a) Translates each SALT specification into an LTL formula and then into a Büchi automaton. Finally determines the language accepted by the intersected automata. (b) First generates the AND-connected SALT specification, which in turn is translated into LTL and then into a Büchi automaton whose accepting language is checked.

The current implementation follows the steps depicted in Figure 5.2b. A new SALT specification is generated that is the conjunction of specifications $\phi_1$ to $\phi_n$. The SALT compiler[1] generates LTL formula $\phi'$ and the tool LTL2BA[2] [80] by Oddoux and Gastin is used to translate the LTL formula into a Büchi automaton. Next, it is checked whether $\mathcal{L}(\mathcal{B})$ is empty or not. If $\mathcal{L}(\mathcal{B})$ is empty, we know that the set of initial SALT specifications is inconsistent, since there is obviously no accepting automaton run.

In Figure 5.3 the AND-connected SALT specification of the running example is given.

[1]http://salt.in.tum.de/
[2]http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php

```
assert (always (
  (always ("sdc_active")) and
  (always ("driver_left"  implies (eventually "car_left"))) and
  (always ("driver_right" implies (eventually "car_right"))) and
  (always ("driver_sdc"   implies (eventually "sdc_active"))) and
  (never  ("driver_left"  and "driver_right")) and
  (never  ("sdc_active")  between
          incl req        ("driver_left" or "driver_right"),
          excl req not    ("driver_left" or "driver_right")) and
  (always ("sdc_active")  between
          incl req        ("search_active"),
          excl req not    ("search_active"))
))
```

**Figure 5.3.:** Conjunction of the presented SALT specifications.

### 5.2.3. Discussion

A prototype of the presented method is implemented within the COLA engineering environment. This initial work, however, has the potential for future extensions. Besides an extension to TLTL, most aspects concern usability in particular the user interface. These are:

(i) As mentioned before, features offer a specific service that interacts with a subset of the complete system interface. Hence, variables used within a SALT specification can only refer to port names of that sub-interface. One extension would be to guide the developer by only allowing variables referring to port names.

(ii) As specifications follow in most cases specific patterns—as described by Dwyer et al. [64]—such a pattern-based user guidance could simplify editing of specifications and hence reduce the risk of potential erroneous specifications. Campetelli et al. [40] pursue a similar objective within AUTOFOCUS.

## 5.3. Deterministic Models

### 5.3.1. Introduction

In the context of automatic deployment of COLA models onto a distributed target hardware platform, it is highly important that the deployed system behaves deterministically. This requires that the possibly non-deterministic source model of the Logical Architecture is transferred into a deterministic runnable entity. During the deployment steps reported in Section 4.4, which fully automatically generate from a COLA model an executable system for the introduced hardware platform, C code generation is involved. The central and core idea of the development process is to take the steps from model level to implementation level while preserving the COLA semantics as close as possible. For this reason, randomness is not a desired system characteristic. As mentioned in Section 4.2.2, it is of course possible to generate deterministic code from non-deterministic models. In this case, code generators need some deterministic rules to resolve the non-determinism (cf. also MATLAB/Stateflow's 12 o'clock semantics). To be semantics preserving down to the execution platform, all involved process steps have to be semantics preserving in general and deterministic in particular. When having a closer look on the COLA modelling language, one notices at a glance that non-deterministic system behaviour can only occur when using automata. Underspecification and thus non-determinism might be a desired characteristic on the Feature Architecture. Underspecification is oftentimes used if some aspects of a system behaviour are not yet defined and thus left open intentionally.

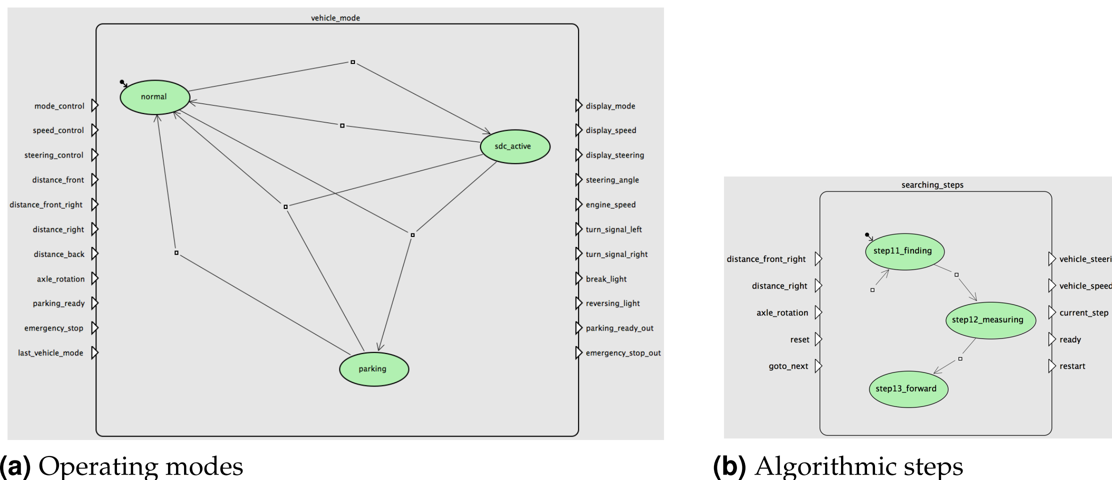
Note, currently there is the restriction that COLA models on level of the Logical Architecture have to be deterministic when generating executable C code. This circumstance is due to the fact that the implemented code generator [90] does not have any rules to resolve non-determinism and COLA's semantics is not guided by a 12 o'clock semantics, for example. Hence, it is all the more important that engineers are made aware of possible problems. Obviously, a non-deterministic source model could possibly result in different source codes, when running the generator several times. In consequence, the resulting target system might behave in an unexpected way. Besides the obvious reason, a pragmatic but not to undervalue reason—especially in the field of safety-critical embedded systems—is tool qualification: in the context of avionics systems—since COLA is developed for, but not limited to automotive systems—the standard DO-178B [174] requires thereafter qualified tools to work deterministically, i. e., on the same inputs always result in the same output. Similar standards for the automotive domain include

the international standard IEC 61508 for ‘functional safety of electrical/electronic/programmable electronic safety-related systems’ [102] and the international standard ISO 26262 (‘Road vehicles – Functional safety’) [104].

If the input model, for instance, for the code generation step was non-deterministic, the generated output would not be the same in general. In consequence, COLA models have to be checked whether they are non-deterministic or not and when, why, and where. In the following, the technical realisation and its restriction will be detailed on.

## 5.3.2. Problem

COLA uses the notion of automata to model control-flow and express distinct system behaviour in the case of so-called mode automata (cf. Section 4.3.1 for details). Automata are the only language constructs that enable non-deterministic behaviour. They determine their next state by evaluating from the current state all outgoing transitions (guarded by conditions) under the input vectors present at the automaton’s interface. Depending on the input vectors and the guarded transitions, it is possible that more than one transition is enabled at the same time, hence leading to a set (more than one) of possible next states. As automata are powerful and widely used modelling constructs—be it to define system modes or just for algorithmic reasons—it is worth analysing them. Examples for both mentioned cases are given in Figures 5.4a and 5.4b, respectively. Figure 5.4a




**(a)** Operating modes **(b)** Algorithmic steps

**Figure 5.4.:** Figures (a) and (b) show the usage as mode automaton and as an automaton describing certain steps of an algorithm.

depicts the basic distinction between the different operating modes of one of

the case studies: 'normal', 'sdc_active', and 'parking'. In the second example, depicted in Figure 5.4b, different steps within the parking procedure are shown. The differentiation between both cases, however, is in many cases fluent. The second example could also be interpreted as modes, however, to regard them as a part or as a step of an algorithmic description seems more appropriate in this case. The automaton of the first example would be non-deterministic if for example there were input values that simultaneously activate the transitions from 'normal' to 'sdc_active' and 'parking'. Therefore, the involved guards have to evaluate to **true**. Tool support hinting the developer to potential problems would be an important brick in the set of quality-increasing activities.

In Figure 5.5 an example automaton is depicted. It consists of three states and two transitions. Each transition is guarded by an expression over $x$, which—of course—is also part of the automaton's input interface. The variable $x$ is assumed to be a natural number, i. e., $x \in \mathbb{N}$. Beginning with state 'State$_1$', the two successive states 'State$_2$' and 'State$_3$' are reachable, depending on the value of $x$. If formula

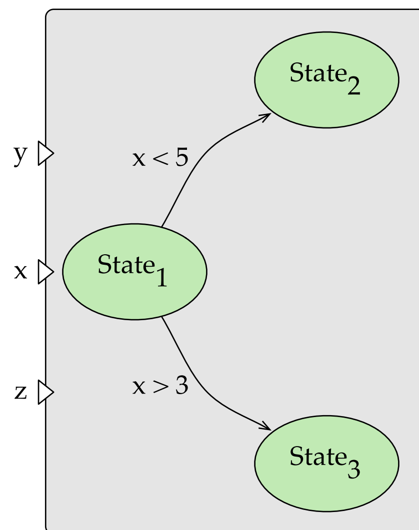



**Figure 5.5.:** For $x = 4$ both transitions are enabled simultaneously.

$\varphi \equiv (x < 5) \wedge (x > 3)$, which is the conjunction of both guarded predicates, has a satisfying assignment for variable $x$, then there is a possible non-deterministic situation. Both guarded transitions are enabled in this case. At a first glance, one can easily see that both conditions $x < 5$ and $x > 3$ evaluate to **true** for $x = 4$ and thus $\varphi$ is satisfiable. Of course, this is *locally* true in the example. However, it might be possible that $x$ cannot be equal to $4$ for some reason. Therefore, the obtained result is an *over-approximation*, since false positives are possible.

Engineers, however, can be made aware of a possible problem.

Figure 5.6 gives the example where $x = 4$ can never occur. Here, the original

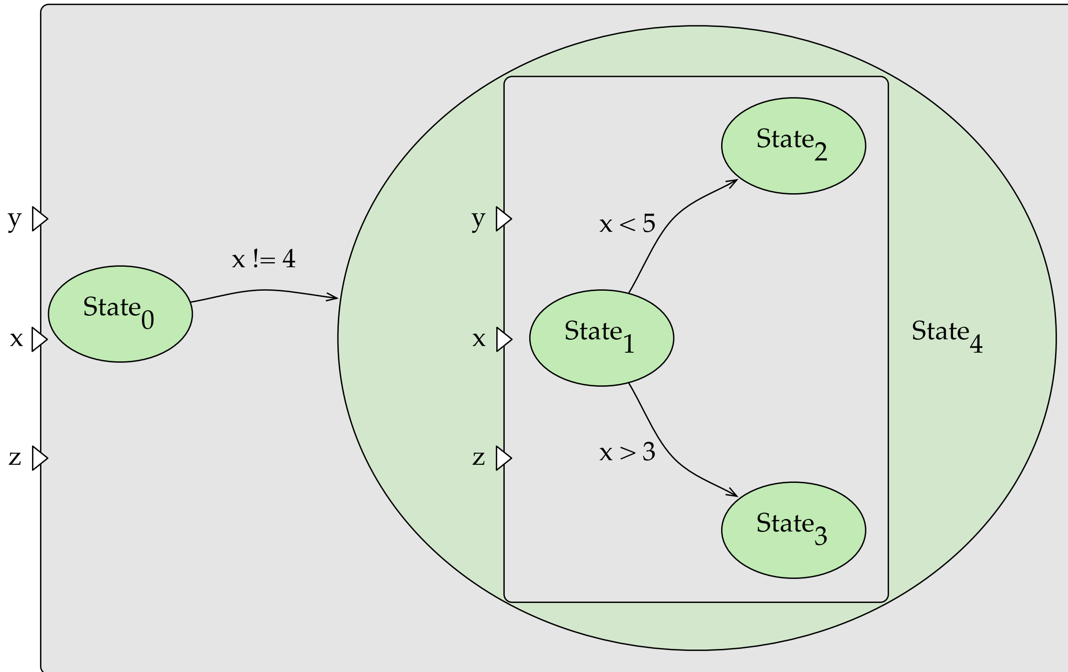



**Figure 5.6.:** The embedded automaton is deterministic, since it is only executed if $x \neq 4$.

automaton is embedded into a state 'State$_4$' of another automaton. 'State$_4$' is only reachable from 'State$_0$' if the condition $x \neq 4$ holds. At this point, one can see that the embedded automaton is deterministic although it does not look like at the first sight. Therefore, the environment or its context has to be considered in order to obtain better results. However, if the automaton was not embedded hierarchically, but some computations were done before, as illustrated in Figure 5.7, the situation would have been much more difficult. In the given example, one has to check whether it is possible that port $x$ can ever hold the value 4. Only in this case a non-deterministic behaviour can occur. Again, one has to check whether it is possible that $M$ eventually provides $x = 4$. In general, this is a very hard question due to the Turing-completeness of COLA and in general even undecidable. Techniques known from program analysis and model checking have to be applied in order to try to compute for the example at hand those values $x$ can take.

Suppose $M$ is restricted, i. e., we do not allow feedback loops and automata, then a new predicate encoding the impact of $M$ on the variable $x$ can easily be derived. In this case, exemplary depicted in Figure 5.8, one can derive the new

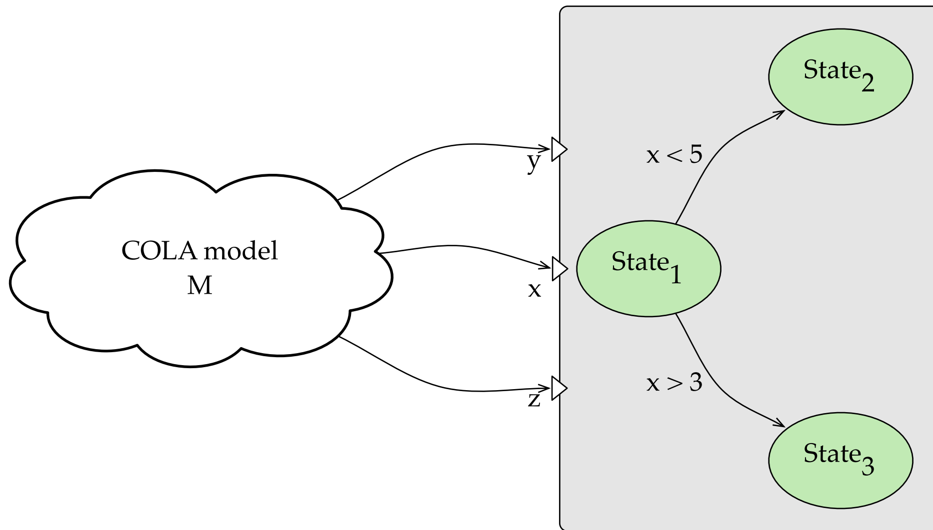



**Figure 5.7.:** COLA model $M$ has to be checked in order to know whether port $x$ can ever hold the value 4.

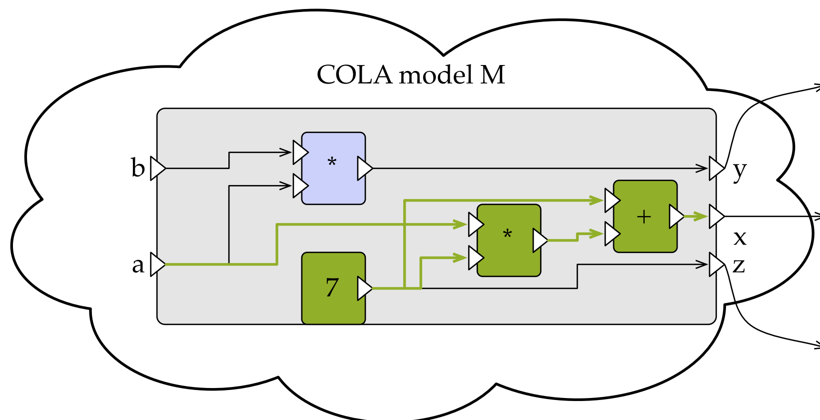



**Figure 5.8.:** Pruning the search space by deriving new predicates by backwards search. In the example: $x = 7a + 7$.

predicate $x = 7a + 7$. All together we gain:

$$\varphi \equiv (x < 5) \wedge (x > 3) \wedge (\boldsymbol{x = 7a + 7}) \tag{5.1}$$

If there is an assignment to the variables $x$ and $a$ satisfying $\varphi$, then $M$ does not change the non-determinism of the automaton. Hence, the backward traversal has to be continued starting at port $a$ until sources are reached or, if like in the given example, $\varphi$ cannot be satisfied anymore. Now, we know that $x$ cannot be equal to 4 because there is no value for $a$ satisfying the equation $4 = 7a + 7$

on natural numbers. If, however, $a$ had been a rational number, $\varphi$ would have been satisfiable with $\beta(x) = 4$ and $\beta(a) = -\frac{3}{7}$. In this thesis, $\beta$ denotes a function returning the assignment of a variable, e. g. $\beta(x) = 4$. Let $\boldsymbol{\beta}(\varphi)$ denote the set of variable assignments satisfying $\varphi$. In this case, the backwards search has to be continued at port $a$. Again, the above discussed cases that can occur, namely, model $M$ is the implementation of an automaton state, or $M$ is part of a larger network.

### 5.3.3. Realisation

Coming back to the example in Figure 5.5. Starting from 'State$_1$', there is a non-deterministic behaviour if guards $x < 5$ and $x > 3$ are satisfied simultaneously. Therefore

$$(x < 5) \wedge (x > 3) \tag{5.2}$$

has to be satisfied. This is true for $x = 4$.

Generally, each automaton state having more than one next state is a potential candidate for non-deterministic behaviour.

**Definition 3** (Signature of a COLA unit [130])**.** *Let $P_{in} = \langle a_1 : t_1, \ldots, a_k : t_k \rangle$ and $P_{out} = \langle a_{k+1} : t_{k+1}, \ldots, a_n : t_n \rangle$ with $k, n \in \mathbb{N}$ be (ordered) lists of typed ports, such that all port identifiers $a_1, \ldots, a_n$ are pairwise disjoint. A typed port is denoted as $a_j : t_j$, i. e., a port identifier $a_j$ with a type $t_j$, $1 \leq j \leq n$. A signature is written as $\sigma = (P_{in} \rightarrowtail P_{out})$.*

**Definition 4** (Automaton)**.** *Let $\mathcal{A} = \langle Q, P, \rightarrow, q_0, \sigma \rangle$ be a* COLA *automaton. $Q$ is the finite set of states, $P$ a finite set of guard predicates, and $\rightarrow \subseteq Q \times P \times Q$ is a ternary relation of guarded transitions. $q_0$ denotes the initial state and $\sigma$ the interface of the automaton. If $p, q \in Q$ and $l \in P$, then $(p, l, q)$ or $p \xrightarrow{l} q$ denotes a transition from $p$ to $q$, which is taken if and only if the predicate $l$ evaluates to* **true** *under the values present at the input side $P_{in}$ of the interface.*

Algorithm 1, which checks whether a given automaton is deterministic or not, is explained in the following. For each state of the automaton under consideration, all pairs (combined with AND) of outgoing guarded transitions ($gs$) are combined with OR (cf. line 4) yielding $\varphi$. Next it is checked whether there exists an assignment $\boldsymbol{\beta}$ satisfying $\varphi$ (cf. line 5), i. e., there are input values simultaneously satisfying at least two outgoing transitions. In this case, the automaton is locally non-deterministic; otherwise, the automaton is deterministic in any case. Next, the **while**-loop ascends the model as shown in the scenario of Figure 5.6. Here the automaton at hand is embedded as the behavioural state specification of a further

```
input  : Automaton 𝒜 = ⟨Q, P, →, q_0, σ⟩ to check whether it is
         deterministic or not
output : Assignments (values) β for input ports in the case of
         non-determinism, otherwise nil

1  φ ← nil
2  foreach p ∈ Q do
3      G = {g ∈ P | (p, g, q) ∈→}
4      φ ← φ ∨ ⋁_{g1,g2∈G, g1≠g2} (g1 ∧ g2)      /* assuming g1 ∧ g2 ≡ g2 ∧ g1 */
5  if β(φ) ≠ ∅ then
6      ℬ ← 𝒜
7      while ¬ (reachedTopLevel() ∨ β(φ) = ∅) do
8          Let 𝒞 = ⟨Q', P', →', q'_0, σ'⟩ be the automaton having ℬ as an
           implementation of one of its states q' ∈ Q'
9          G' = {g' ∈ P' | (p', g', q') ∈ →'}
10         φ ← φ ∧ (⋁_{g'∈G'} g')
11         ℬ ← 𝒞
12     if β(φ) ≠ ∅ then
13         return β(φ)
14     else
15         return nil
16 else
17     return nil
```

**Algorithm 1:** Checks whether a given automaton $\mathcal{A}$ is deterministic or not. In the positive case it returns $nil$, otherwise it returns a variable assignments (counterexample) $\beta$ as proof for its local non-determinism.

embracing automaton. It is possible that this state is reachable from multiple predecessor states. Thus, the respective guarded transition conditions have to be combined with OR (cf. line 10) and added (AND) to $\varphi$. Next, the hierarchical COLA model is ascended until either $\varphi$ is no more satisfiable or the top level has been reached (cf. `reachedTopLevel()` in line 7). Now, if $\varphi$ is still satisfiable, then there is a strong evidence that the automaton is non-deterministic—at least at model level. Why this restriction? If for instance the distance to an ahead driving car is modelled using a source block on level of the Logical Architecture, the engineer may use the integer data type. Suppose the outlined algorithm states that some interesting automaton may be non-deterministic for an input parameter 'distance' with the value say $-10$. In a technical realisation, however, a distance will never have a negative value—assuming no hardware malfunction—and thus the provided *counterexample* is spurious. Hence, the gained input assignments are always *over-approximations* unless variables are traced back to constant COLA blocks only.

The current implementation uses the YICES [63] SMT-solver as backend to solve $\varphi$. Indeed, any other solver would have been applicable, too. As YICES is not capable to work with non-linear arithmetic, predicates like those for $y$ in Figure 5.8, namely $y = ab$, are not possible. In such a case a solver for non-linear arithmetic as for instance the ABsolver by Bauer et al. [20] is necessary.

Figure 5.9 shows the integration into the COLA engineering environment. In this example, the non-deterministic behaviour was introduced by hand, for demonstration purposes. Here, the two predicates

$$g_1 \equiv (mode_control = 0) \vee emergency_stop \tag{5.3}$$
$$g_2 \equiv (mode_control = 0) \wedge (\neg emergency_stop) \wedge (steering_control = 0) \tag{5.4}$$

determine $\varphi \equiv g_1 \wedge g_2$. In this formula, only Boolean and integer variables occur. By construction, $\varphi$ is satisfiable with the assignment

$$\beta(\varphi) = \{emergency_stop \leftarrow \textbf{false}, steering_control \leftarrow 0, mode_control \leftarrow 0\} \tag{5.5}$$

The example does not use hierarchically nested automata, thus Algorithm 1 does not enter the **while**-loop. As the variables occurring in $\varphi$ are influenced by source blocks, the given example indicating non-deterministic behaviour may be spurious. Hence, the engineer has to verify whether the provided result makes sense. A negative distance in the example above obviously makes no sense. The variable assignment given in Formula (5.5) is not spurious.

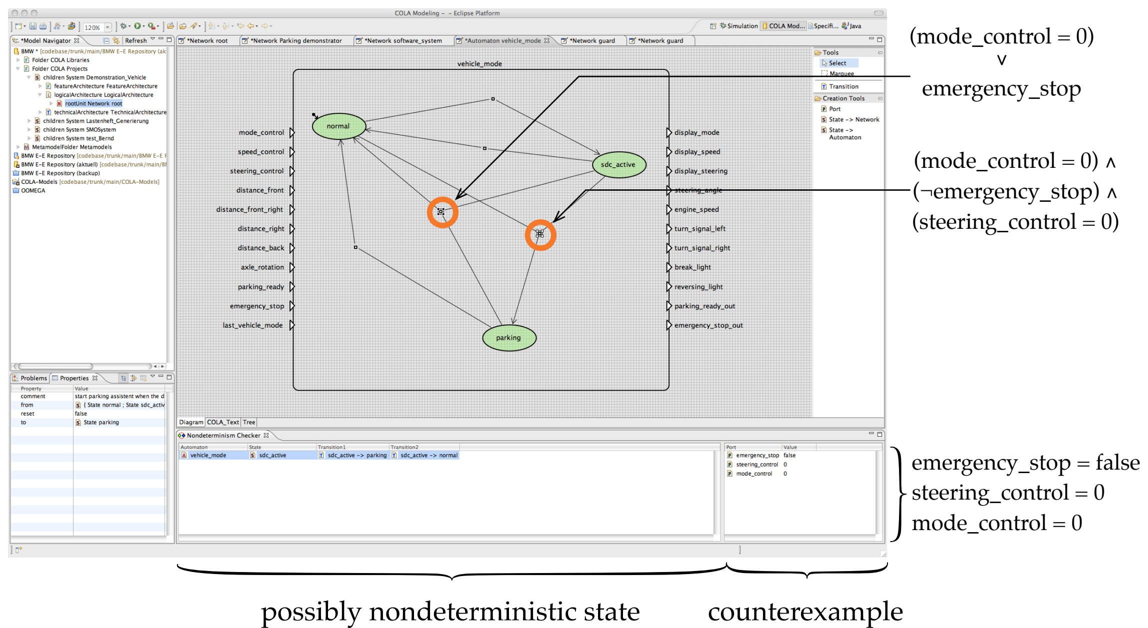



**Figure 5.9.:** Verification result visualised within the COLA-IDE.

### Complexity Considerations

Initially, the length of $\varphi$ for a given automaton $\mathcal{A} = \langle Q, P, \rightarrow, q_0, \sigma \rangle$ is determined by the number of contained states $|Q|$ and the length of the guarded transitions. For each state of the automaton and for each pair of outgoing transitions $g_1, g_2$, possibly $\frac{\binom{|Q|-1}{2}}{2}$ many, a clause $(g_1 \wedge g_2)$ is added. During each **while**-loop pass, the length is extended by a clause of length constant in $|Q'| - 1$. The number of loops is bounded by the embedding-depth $\delta$ of $\mathcal{A}$. As COLA models are finite, termination of Algorithm 1 is guaranteed. For a flat automaton, $\delta = 0$ as in the example depicted in Figure 5.9.

When using a technique similarly to *iterative SAT solving* [71], but in contrast with SMT formulae, the solver has to be called only once with the initial formula or set of assumptions $\varphi$. In the following execution, only $\delta$-times new assertions are added, which makes the approach feasible even for larger models.

## 5.4. COLA Model Analysis via a Translation to Coloured Petri Nets

Modelling formalisms with a well-defined formal basis including syntax and semantics facilitate the use of formal methods for quality improving activities. Testing, simulation, and verification are only some of them.

> The Heart and Soul of Model-Driven Software Development",

as Sendall and Kozaczynski [182] characterise *model transformation*, is a further category. This section introduces a transformation schema from COLA core to Coloured Petri nets (CPNs) [107–109] illustrating one of the benefits of having a formal foundation. CPNs are an example for a modelling language with an exact mathematical definition of their execution semantics. Coloured Petri nets are an extension of classical Petri nets and facilitate hierarchical system modelling and distinguish between tokens having a value of a certain type. A transformation between two formalisms allows mutually usage of available analysis tools and techniques—assumed semantics preservation during translation. In the presented case, the power of the *CPN Tools* [50] is used to analyse COLA models. Of course, the analysis techniques of the used CPN tools could also have been redeveloped and integrated into the COLA-IDE. However, this was not the intention of the research question and thus has not been realised within the initial COLA breakthrough. Nevertheless, in the following, a translation schema from COLA core to CPN is developed and can be considered as a proof of concept in the sense of rapid prototyping. After a brief introduction into Coloured Petri nets in the following section, the actual translation scheme is presented.

### 5.4.1. Introduction to Coloured Petri Nets

Coloured Petri nets—similar to COLA—are a graphical modelling language emerged from the combination of Petri nets [168] and the functional programming language Standard ML (SML) [158, 184]. They are also referred to as high-level Petri nets. In order to be able to cope with the normally large size of real life systems and to introduce a better system overview, CPNs offer the possibility of hierarchically modelling, i. e., parts of the model are combined into submodules. On the one hand, the usage of Petri nets offers an effective framework for *modelling* concurrency, communication, and synchronisation. On the other hand, the application of SML facilitates the *definition* and *manipulation* of the data.

The application within the presented work focuses only on the analysis techniques, since the COLA-IDE already features powerful modelling capabilities. The *CPN Tools* [50] framework integrates modelling, simulation, and analysis capabilities and has been successfully applied in various application areas and industry projects [109, 125, 126, 159, 164].

The following definitions of hierarchical and non-hierarchical CPNs should serve to easier understand the terminology used for the translation of COLA models. We allow colour sets to be multi-sets, i. e., they are sets allowing multiple appearances of the same element. For better readability multi-sets are marked with the subscript *MS*. For more detailed and complete definitions see [107–109].

**Definition 5** (Non-hierarchical CPN (cf. [108]))**.** *A non-hierarchical CPN is a 9-tuple* $\Omega = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$ *with:*

*(i)* *A finite set of* places $P$ *and* transitions $T$ *such that* $P \cap T = \varnothing$.

*(ii)* *A set of directed* arcs $A \subseteq (P \times T) \cup (T \times P)$.

*(iii)* *A finite set of* colour sets $\Sigma$.

*(iv)* *A finite set of* variables $V$, $\mathsf{type}(v) \in \Sigma, \forall v \in V$.

*(v)* *A* colour set function $C : P \rightarrow \Sigma$, $C(p) \in \Sigma, \forall p \in P$.

*(vi)* *A* guard function $G : T \rightarrow \mathit{Expr}$, $\mathsf{type}(G(t)) = \mathbb{B}, \forall t \in T$.

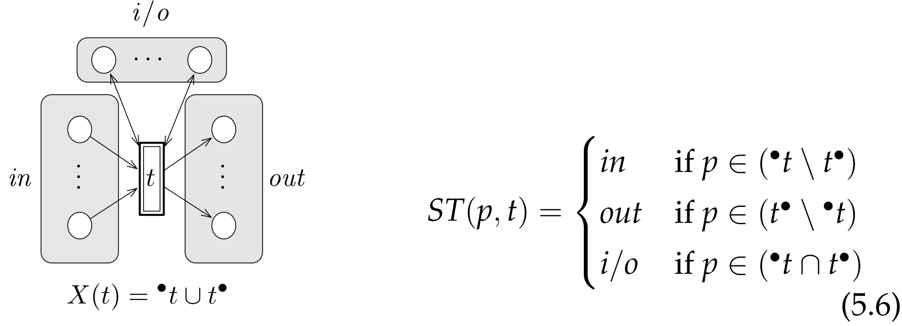
*(vii)* *An* arc expression function $E : A \rightarrow \mathit{Expr}$, $\mathsf{type}(E(a)) = C(p)_{MS}, \forall a \in A$ *and* $a$ *is connected to* $p \in P$.

*(viii)* *An* initialisation function $I : P \rightarrow \mathit{Expr}$, $\mathsf{type}(I(p)) = C(p)_{MS}, \forall p \in P$.

As Coloured Petri nets are an extension to Petri nets without colours, they also consist of places $P$ and transitions $T$ connected by arcs $A$ but moreover each place $p \in P$ can hold a certain colour, referred to as colour set $C(p) \in \Sigma$. CPNs can also use variables $v \in V$ of a type, i. e., $\mathsf{type}(v) \in \Sigma$. Transitions $t \in T$ can be guarded by an expression *Expr* which shall evaluate to either `true` or `false`, i. e., $\mathsf{type}(G(t)) = \mathbb{B}$. The arc expression function $E$ maps each $a \in A$ into an expression of type $C(p)_{MS}$ where $p \in P$ is that place connected to $a$. Finally, places may be initialised with an expression $\mathit{Expr}$ without variables.

In the following, the notion of *substitution transitions* is used. A substitution transition is used to hierarchically decompose CPNs, i. e., they represent a more detailed submodule of the CPN. The set of places belonging to the preset and the

postset of a transition $t$ is denoted $X(t) = {}^\bullet t \cup t^\bullet$, respectively. *Socket nodes* are called the places $p$ surrounding a substitution transition $t$, i. e., $p \in X(t)$. The *socket type* function $ST$ is defined as follows (cf. [108]):




$$ST(p,t) = \begin{cases} in & \text{if } p \in ({}^\bullet t \setminus t^\bullet) \\ out & \text{if } p \in (t^\bullet \setminus {}^\bullet t) \\ i/o & \text{if } p \in ({}^\bullet t \cap t^\bullet) \end{cases} \tag{5.6}$$

In the figure, the notion of a substitution transition and that of socket nodes and types, respectively, is visualised. This parenthesis was important for the following definition of hierarchical Coloured Petri nets.

**Definition 6** (Hierarchical CPN (cf. [108]))**.** *A hierarchical CPN is a 9-tuple* $\Omega_H = \langle S, SN, SA, PN, PT, PA, FS, FT, PP \rangle$ *with:*

- *A finite set of* pages $S$*. Each page is a non-hierarchical CPN:* $\forall s \in S$, $s = \langle P_s, T_s, A_s, \Sigma_s, V_s, C_s, G_s, E_s, I_s \rangle$*, the set of net elements of each page pair are disjoint.*

- *A set of* substitution nodes $SN \subseteq T$*, where* $T = \bigcup_{s \in S} T_s$ *is the set of transition of the entire CPN.*

- *A* page assignment function $SA : SN \rightarrow S$*, such that no page is a subpage of itself.*

- *A set of* port nodes $PN \subseteq P$ *where* $P = \bigcup_{s \in S}$ *is the set of all places of the entire CPN.*

- *A* port type function $PT : PN \rightarrow \{in, out, i/o, general\}$*.*

- *A* port assignment function $PA : SN \rightarrow 2^{X(SN) \times PN}$ *such that:*
  - *The relation between socket nodes and port nodes is defined as follow:* $\forall t \in SN : PA(t) \subseteq X(t) \times PN_{SA(t)}$.
  - *Correct types for socket nodes are required:* $\forall t \in SN, \forall (p_1, p_2) \in PA(t) : [PT(p_2) \neq general \Rightarrow ST(p_1, t) = PT(p_2)]$.
  - *Related nodes have the same colour set and initialisation:* $\forall t \in SN, \forall (p_1, p_2) \in PA(t) : [C(p_1) = C(p_2) \wedge I(p_1)\langle\rangle = I(p_2)\langle\rangle]$.

- *A finite set of* fusion sets $FS \subseteq P_s$*, such that all elements have the same colour set and equivalent initialisation expressions.*
- *A* fusion type function $FT : FS \rightarrow \{global, page, instance\}$*, such that page and instance fusion sets belong to a single page.*
- *A multi-set of* prime pages $PP \in S_{MS}$.

A hierarchical Petri net consists of a finite set of pages $S$ that are associated to substitution nodes $SN$, i. e., substitution transitions, using the page assignment function $SA$. Each page is a non-hierarchical Petri net. Moreover, the interaction between the high-level substitution transition and its associated page is realised using the port assignment function $PA$. It relates the socket ports, i. e., those places $X(t)$ surrounding the substitution transition $t$ with port nodes of the corresponding direct subpage. These ports have to have the same port type—determined using $PT$.

### 5.4.2. Translation Schema

In the following, a translation schema from COLA core to CPNs is proposed. Therefore, each language construct is translated one after another. Beginning with basic units, namely functional blocks, stepwise more and more complex translation schemas for units like networks and automata are given.

For the translation both hierarchical and non-hierarchical CPNs are used. Units that can be decomposed are translated into hierarchical CPNs, those that cannot into non-hierarchical CPNs. In order to make sure that no value is written into a non-empty place, input and output places (and where necessary) are defined as lists of a given data type. Thus, apart from other constraints, each transition connected to such places fires only if its postset is empty. This reflects the semantics defined in COLA.

Before specifying the translation schema, the following function is defined first: $\pi : \mathsf{io}(\sigma) \rightarrow P$, which maps the set of COLA input $in(\sigma)$ and output $out(\sigma)$ ports into the set of CPN places, with $\mathsf{io}(\sigma) = in(\sigma) \cup out(\sigma)$. A unit's signature $\sigma$ is specified as follows: $\sigma = (\langle lop : tt, rop : tt\rangle \rightarrowtail \langle result : m\rangle)$ (cf. also Definition 3).

#### Constant Blocks and Delays

For constant blocks and delay units the translation is straightforward: *constant blocks* are translated into a single CPN place initialised with the corresponding value. For each input and output of a *delay*, a separate CPN place as well as a

transition to connect them is generated. A translation of a delay, e.g. `pre_1`, can be found in Figure 5.16a. Note that the delay is modelled as a substitution transition whose subpage is depicted in Figure 5.10. On the superpage, a transition (`pre_1` in the example) with two places connected with arcs is created. The behavioural translation is done in the subpage depicted in Figure 5.10 not visible in Figure 5.16a. The delay's initialisation—1 in the example—is reflected with the marking `1‘[1]` at place `init`. Keeping the same structure as the original COLA model should serve for a better understanding of the translation process.

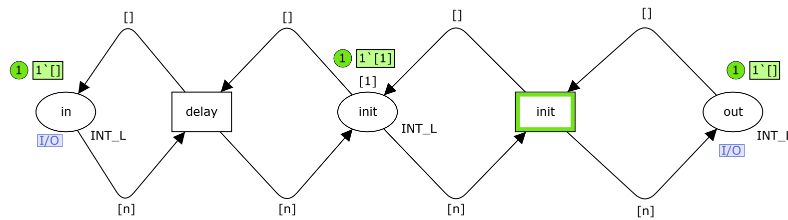
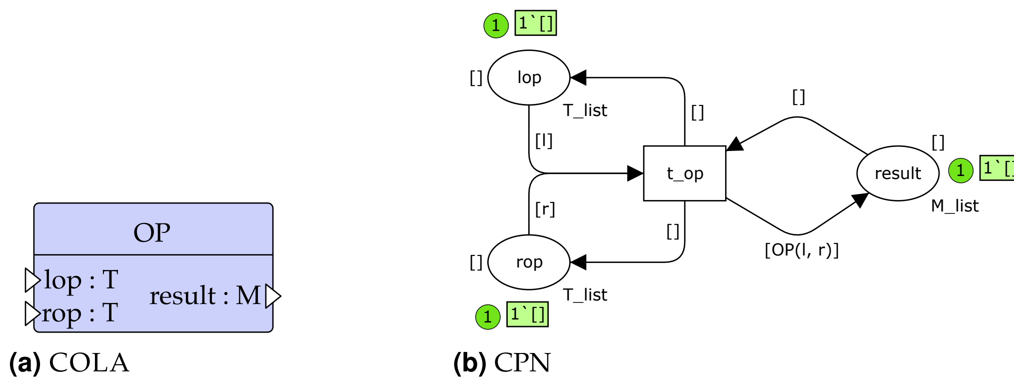



**Figure 5.10.:** CPN for a COLA delay block initialised with 1.

### Functional Block




**(a)** COLA **(b)** CPN

**Figure 5.11.:** (a) COLA basic block with two input ports of type $T$ and an output port of type $M$. (b) Corresponding CPN with three places and one transition.

In Figure 5.11 a COLA functional block and its translation is depicted. The translation schema for a functional block is defined in Algorithm 2. Since functional blocks cannot be further decomposed, their translation is straightforward. Input and output ports are transformed into CPN places ($P$), including their corresponding data types (C). A transition ($T$) is generated to reflect the operation OP and

is accordingly connected to places by arcs ($A$). Arc inscriptions ($E$) matching the empty list [ ] play a key role for the generated CPN model. On the one hand, they force the transition to fire only if its postset is empty, cf. $(result, t_{op})$ in Figure 5.11b. In this way the behaviour defined in COLA is reflected, i. e., no new value is added to an output port unless old values are consumed. On the other hand, they notify other modules connected to them that the data residing in the input ports has been consumed (cf. the outgoing arcs from $t_{op}$ to the left), i. e., new values can proceed. To achieve this, lists of used data types ($\Sigma$) are defined. Variables ($V$) corresponding to a data type are used to read the input and process the data according to the operation OP, cf. the arc inscription of $(t_{op}, result)$ in Figure 5.11b. The guard ($G$) of the transition is always **true**. All places are initialised ($I$) with the empty list.

**input** : COLA functional block
$\mathsf{FB} = \langle n, \sigma = (\langle lop : tt, rop : tt\rangle \rightarrowtail \langle result : m\rangle), \mathsf{OP}\rangle$
**output**: CPN $cpn = (P, T, A, \Sigma, V, C, G, E, I)$

1 $P = \pi(\mathsf{io}(\sigma))$, i.e. $\{lop, rop\} \cup \{result\}$
2 $T = \{t_{op}\}$
3 $A = \{(lop, t_{op}), (rop, t_{op}), (t_{op}, result), (t_{op}, lop), (t_{op}, rop), (result, t_{op})\}$
4 $\Sigma = \{tt_l, m_l\}$, $tt_l$ and $m_l$ are lists of type $tt$ and $m$, resp.
5 $V = \{l : tt, r : tt\}$
6 $C(p) = \begin{cases} tt_l & \text{if } p \in \{lop, rop\} \\ m_l & \text{if } p \in \{result\} \end{cases}$
7 $G(t) = \mathbf{true}, \forall\ t\ \in\ T$
8 $E(a) = \begin{cases} [l] & \text{if } a = (lop, t_{op}) \\ [r] & \text{if } a = (rop, t_{op}) \\ [\mathsf{OP}(l, r)] & \text{if } a = (t_{op}, result) \\ [\,] & \text{if } a \in \{(t_{op}, lop), (t_{op}, rop), (result, t_{op})\} \end{cases}$
9 $I(p) = [\,], \forall p \in P$

**Algorithm 2:** Translates a COLA functional block into a corresponding CPN according to the given schema.

### Network

Usually, COLA networks consist of a larger number of subunits connected by channels. An example is given in Figure 5.12. In the following, only the translation of the highest level is given, lower levels are translated analogously. The translation schema for COLA data-flow networks is defined as shown in Algorithm 3.

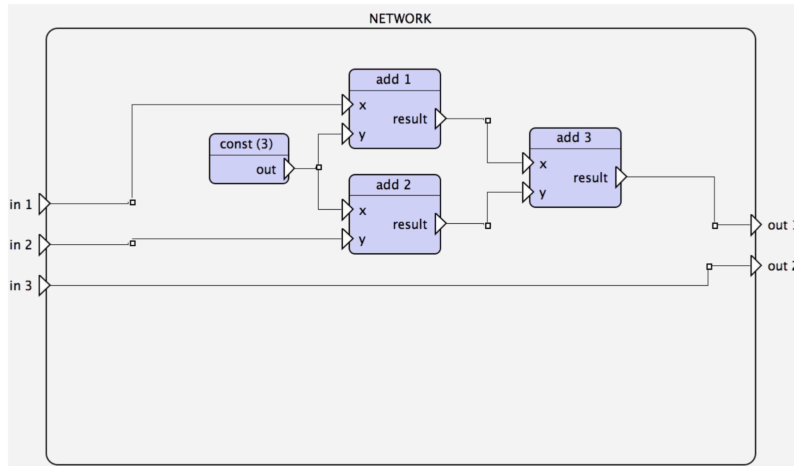
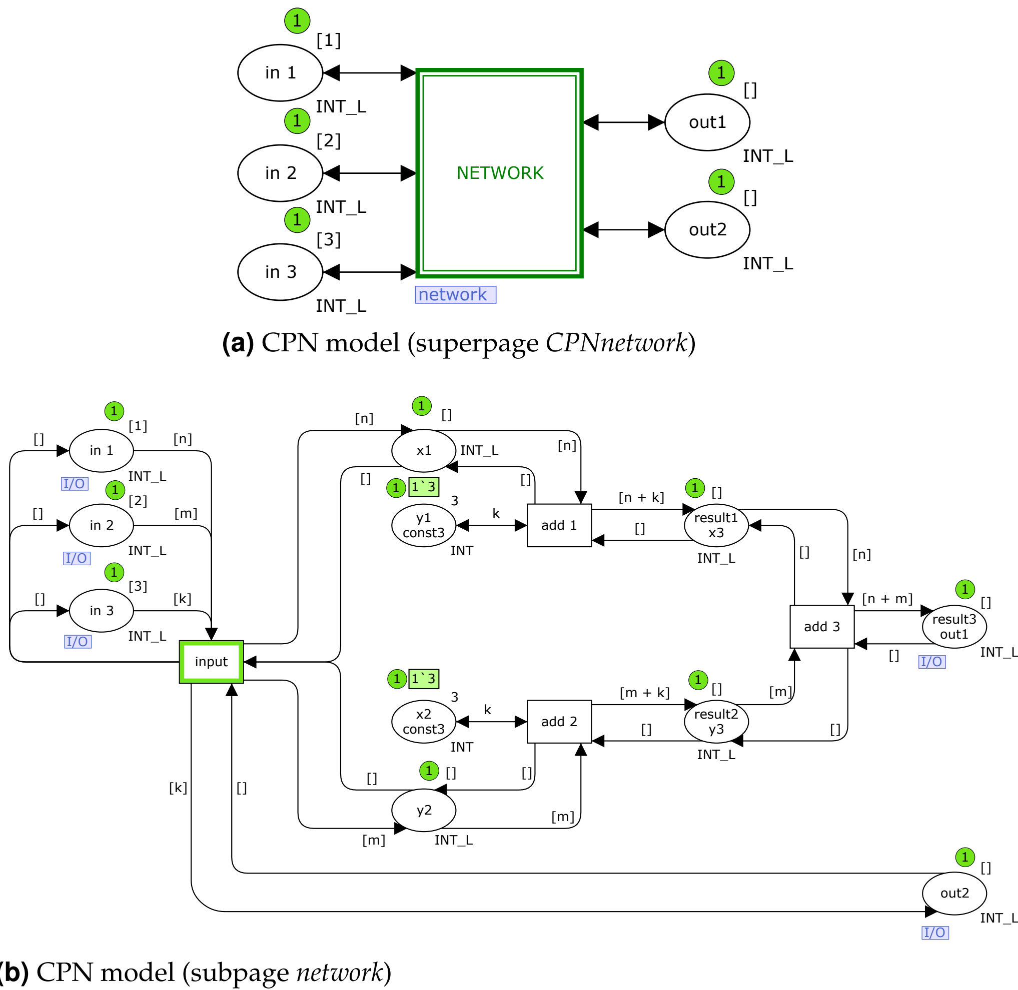



**Figure 5.12.:** Exemplary COLA network.
out 1(in 1, in 2, in 3) = (in 1 + 3) + (in 2 + 3)
out 2(in 1, in 2, in 3) = in 3

**Definition 7** (COLA network [130])**.** *A* network *is a unit* $\langle n, \sigma, I\rangle$ *with a name* $n$*, a signature* $\sigma$ *(cf. also Definition 3) and an implementation* $I = \langle U, C\rangle$ *given by the set of contained subunits* $U$ *and the set of channels* $C$ *connecting them.*

Each network is translated into a corresponding hierarchical CPN. For the top level of each COLA network a page, the super page, is generated, e. g. *CPNnetwork* in Figure 5.13a.

In the following we will refer to the COLA and CPN models and their components in Figures 5.12 and 5.13 when necessary to achieve a better understanding of the translation process. The set of other pages, representing the implementation $I = \langle U, C\rangle$ of the network, are included in $S_U$, where $U$ is the set of subunits participating in the network. Each of these units is separately translated corresponding to its schema type. The translation of the set of channels $C$ is not explicitly given. However, they are important for establishing the connectivity between translated components, e. g. if there is a connection/channel from a COLA unit $A$ to a unit




**(a)** CPN model (superpage *CPNnetwork*)

**(b)** CPN model (subpage *network*)

**Figure 5.13.:** (a) depicts the superpage of a CPN model for the respective COLA model given in Figure 5.12. (b) shows the corresponding subpage.

$B$, in the corresponding CPN model the output places of $A$ are correspondingly glued together with the input places of $B$. Each subunit is represented by the set of substitution nodes $SN$, which consist of transitions, e. g. $n_{\mathsf{NET}} =$ NETWORK, and the set of those ($SN_U$) appearing in the subunits in $U$. $SA$ maps each substitution transition to their implementations in the subpages, e. g. transition NETWORK to the subpage *network*. The set of input and output nodes of *CPNnetwork* ($in_1$, $in_2, \ldots$) are unified with those of the subpages $PN_U$ building the set $PN$. Most of port nodes are of type ($PT$) *i/o* as described in the schema. Now we just need to define the assignment ($PA$) of port nodes to socket nodes, e. g. $in_1$ in *network*, denoted $in_1$@*network*, is assigned to $in_1$ in *CPNnetwork* ($in_1$@*CPNnetwork*). This establishes the connection of places on the superpage to places of the corresponding

subpage. Since the nodes in both pages share commonly the same name, the tuple ($out_1@CPNnetwork, result_3out_1@network$) illustrates best such an assignment.

**input** : COLA network $\mathsf{NET} = \langle n, \sigma, \langle U, C \rangle \rangle$
**output**: Hierarchical CPN $Hcpn = (S, SN, SA, PN, PT, FS, FT, PP)$

1. $S = \{CPNnetwork\} \cup S_U$
2. $SN = \{n_{\mathsf{NET}}\} \cup SN_U$, $n_{\mathsf{NET}}$ is the identifier of $\mathsf{NET}$
3. $SA(SN) = \bigcup_{s \in SN} SA(s)$
4. $PN = \pi(\mathsf{io}(\sigma)) \cup PN_U$
5. $PT(p) = \begin{cases} i/o & \text{if } p \in \pi(\mathsf{io}(\sigma)) \\ PT(PN_U) & \text{if } p \in PN_U \end{cases}$
6. $PA(t) = \begin{cases} \{(in_1@CPNnetwork, in_1@network), & \\ (in_2@CPNnetwork, in_2@network), \ldots, & \\ (out_1@CPNnetwork, out_1@network), \ldots\} & \text{if } t = n_{\mathsf{NET}} \\ PA(SN_U) & \text{if } t \in SN_U \end{cases}$
7. $PP = 1`CPNnetwork$

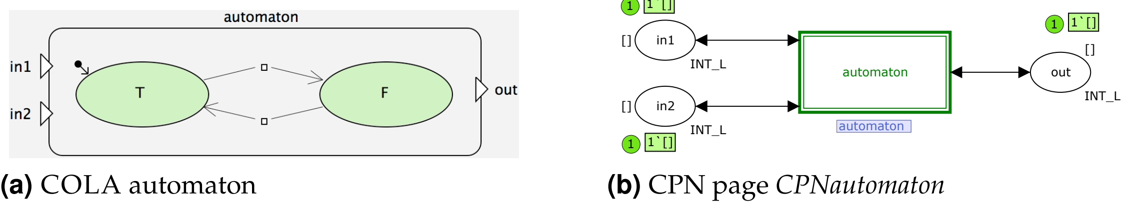
```
/* Note: FS and FT are not considered during the
   translation.                                      */
```

**Algorithm 3:** Translates a COLA network into a corresponding hierarchical CPN according to the given schema.

### Automaton

Automata are the most complex units of COLA. Figure 5.14a shows a COLA automaton and its CPN representation (5.14b). For the translation of an automaton a two-step schema is given (cf. Algorithm 4). The highest level of abstraction is described as a hierarchical CPN in the first step. In the second step the functionality of the automaton, i. e., guard evaluation and state switching, is described as a non-hierarchical CPN. For each state of the automaton, e.g. `T` and `F`, there exists a separate transition, which serves as a substitution transition for the implementation of the underlying network unit (cf. Figure 5.16b). The same figure would represent also the functionality of the automaton in Figure 5.14b, by only replacing `do_nothing` and `working` with `T` and `F`, respectively.

As mentioned and outlined in Algorithm 4, the translation of COLA automata into CPNs is performed in two steps:




**(a)** COLA automaton

**(b)** CPN page *CPNautomaton*

**Figure 5.14.:** In Figure (a) a simple COLA automaton with the two states `T` and `F` is depicted. Its CPN representation can be seen in (b).

(i) The first translation step is similar to the translation of a network, thus we give no further description. We have, however, to stress that the index $Q$ represents the implementation of the underlying network for each automaton state in $Q$. The set of their corresponding substitution transitions is denoted $Q_T$.

(ii) The second step describes by means of non-hierarchical CPNs the next lower level page (*automaton*). Besides the port nodes, determined by $\pi()$, which are needed to be assigned to sockets of the parent or super page (*CPNautomaton*), there are two additional places *State* and *activated* added to the set of places $P$. Place *State* is of type *State* and holds the identifiers of each state in $Q$. *activated* holds the currently active state. The transition *activate State* is responsible for the initialisation of state switching, by feeding the function *state()* with input data and the actual active state. The purpose of function *state()* is to check and control the switching between states, according to the defined guards of the automaton. Let $G = \{g_1, g_2, \ldots, g_n\}$ be the set of the guards of an automaton, $V = \{v_1, v_2, \ldots, v_m\}$ the set of variables used in the guards and $S = \{s_1, s_2, \ldots, s_i\}$ the set of states of the automaton, with $s \in S$. We define the *state*() function and the colour sets *State* and *State_L* as follows:

```
1 fun state(s, v_1, ..., v_m) =
2         if s = s_1 andalso g_1 then s_2
3         else
4         if s = s_2 andalso g_2 then s_3
5         ...
6         else s;
7 colset State   = with s_1 | s_2 | ... | s_i;
8 colset State_L = list State;
```

The rest of the translation schema is straightforward.

### 5.4.3. Translation Algorithm

The idea of the outlined Algorithm 5 is to translate COLA models into CPNs in a DFS manner. For each visited unit, the corresponding translation schema is applied. Once all units are translated there will be loose components and a superfluous number of places (representing each input and output port of each component). To reduce the number of places and establish the corresponding connectivity between components, we glue together input and output places (cf. line 19) regarding the defined channels in the original COLA model, i. e., the corresponding source and destination ports. Finally, to accommodate the structure of the generated hierarchical CPN, the connection between subpages and their parent pages is established by assigning ports to sockets (cf. line 20).

There are two special cases that need to be considered during the translation of a network:

(i) If multiple ports read from one and the same port (cf. port *out* of the constant block in Figure 5.12). In this case, we translate the connection in that way that the source of the channel is translated to as many places as there are destinations (cf. places $y_1 const_3$ and $x_2 const_3$ in Figure 5.13b).

(ii) The input and output of a unit are not connected (cf. Figure 5.15 the implementation of the `do_nothing` state). Therefore, we create a *new* place and connect it with the *input* transition and other transitions accordingly (cf. Figure 5.16c). This is done to make sure that the data flow in the network is not broken, i. e., we want to establish a correct consumption of the input data in order to proceed to the output, as required in COLA.

Note that one can merge the transitions *input* and *out*, thus not needing to add the *new* place at all. The transition *input* can often get merged with other transitions and reduce the size of the net, e. g. one could merge *input* and *add* (cf. Figure 5.16d) and deleting the places *in_1* and *in_2*, without changing the behaviour of the net.

### 5.4.4. Example

In Figure 5.15, a screenshot of the COLA simulation tool is depicted. It shows a high level COLA system consisting basically of two automata (`automaton_1`, `automaton_2`), two input constants with the values 3 and 5 and two delay operators (`pre`, initialised with 1). Each automaton has two states, namely `do_nothing` and `working`. In both cases, `do_nothing` always provides the value 0 as output, concerning the behaviour of `working`, however, both automata show a different

implementation. `automaton_1` performs the subtraction of the values present at the input ports `in_1` and `in_2` (`out` := `in_1` - `in_2`). The state `working` of `automaton_2` adds both input values `in_1` and `in_2`, i. e., `out` := `in_1` + `in_2`. Concerning the state transitions, both automaton share similar conditions de-

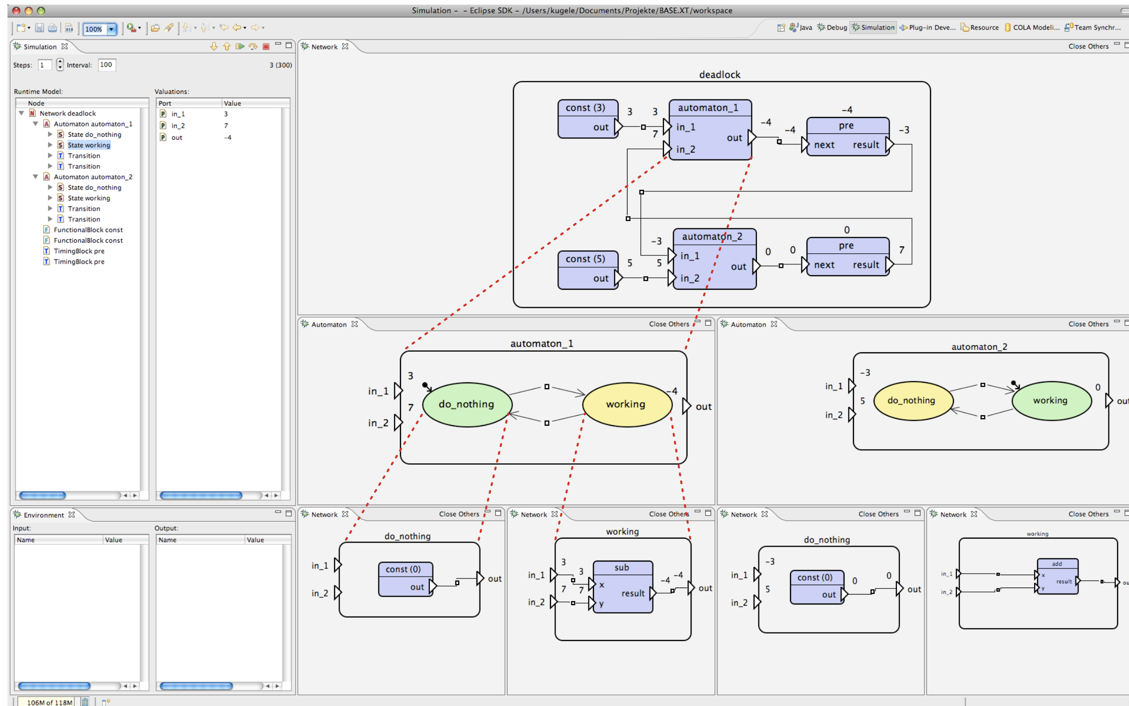



**Figure 5.15.:** COLA simulator: dashed lines are added manually to clarify the hierarchical decomposition.

pending on their varying inputs `in_1` (`automaton_2`) and `in_2` (`automaton_1`), respectively:

$$\texttt{automaton_1: do_nothing} \overset{\texttt{in_2}>0}{\longrightarrow} \texttt{working} \overset{\texttt{in_2}\leq 0}{\longrightarrow} \texttt{do_nothing}$$

$$\texttt{automaton_2: do_nothing} \overset{\texttt{in_1}>0}{\longrightarrow} \texttt{working} \overset{\texttt{in_1}\leq 0}{\longrightarrow} \texttt{do_nothing}$$

If `in_{1,2}` $> 0$, the state changes from `do_Nothing` to `working` and contrariwise if `in_{1,2}` $\leq 0$ the transition from `working` to `do_Nothing` is taken. In all other cases, the automaton stays in the current state.

In COLA a deadlock in a classical sense is not possible. This is due to the fact that the COLA semantics dictates that at each tick of the system execution a new value is assigned to each output port. A deadlock from a Petri net point of view is compared best with a COLA system that is stuck in an automaton state, which

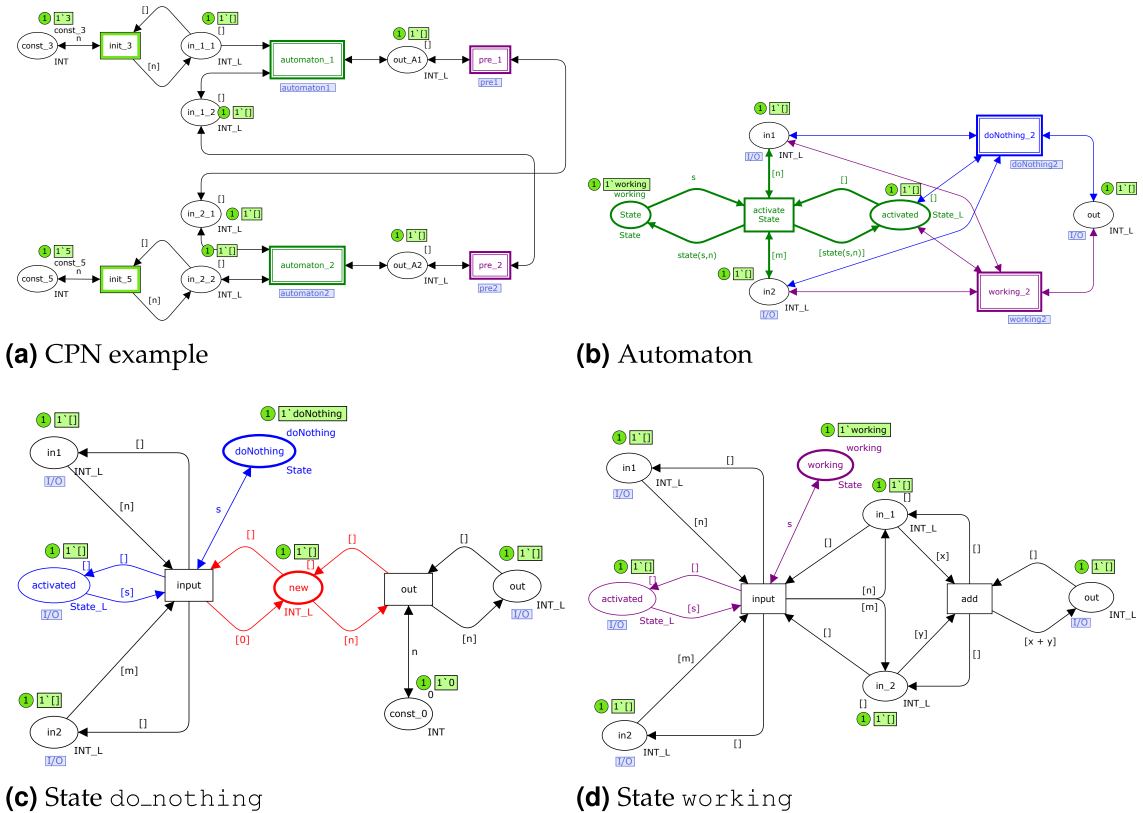
cannot be changed anymore. This might, however, be a system design decision. But in many cases, as in the given example, it is a modelling error. Regarding the example, the values present at the output ports `result` of both delays have a special behaviour: at the first tick both ports emit the value $1$. To simplify matters, we write these values as a result vector $\mathbf{r} = \binom{1}{1}$ where the upper value corresponds to the port value of the upper delay, and the lower value to the lower delay, respectively. The developer has set both values as default for the delays. When considering the behaviour over time we use a matrix-like notation, i. e., the *i*th column of the matrix represents the output values after the *i*th tick. For this simple example, the following infinite sequence

$$\mathbf{M}^{\infty} = \begin{pmatrix} 1 & 2 & -3 & -4 \\ 1 & 6 & 7 & 0 \end{pmatrix} \circ \begin{pmatrix} 0 \\ 0 \end{pmatrix}^{\omega}$$

of port valuations is obtained, i. e., after a finite number of steps (four in this case) the system reaches a deadlock-like state and from then on only emits $\mathbf{r} = \binom{0}{0}$ for $i > 4$ as result. However, for more complex examples, the engineer cannot detect similar behaviour solely using the COLA simulator. Here, the power of the CPN Tools becomes important.

After translating the COLA model into a CPN, using the outlined translation algorithm, the CPNs depicted in Figure 5.16 are obtained. The idea is to automatically construct the state space of the CPN models and finally create the *state space report*, which contains information about standard behaviour properties: dead markings, dead and live transitions, etc. This information collected in the report supports the analysis of a system in an early stage of its development and helps to decreases the number of design errors.

```
Live Transition Instances
--------------------------------
    automaton1'activate State 1
    automaton2'activate State 1
    doNothing1'input 1
    doNothing1'out 1
    doNothing2'input 1
    doNothing2'out 1
    pre1'delay 1
    pre1'init 1
    pre2'delay 1
    pre2'init 1
```




**(a)** CPN example

**(b)** Automaton

**(c)** State do_nothing

**(d)** State working

**Figure 5.16.:** CPN example: (a) The highest abstraction level of the CPN example. (b) Realisation of an automaton. (c) Realisation of the state do_nothing. (d) Realisation of the state working (*automaton_2*).

The expected behaviour is reflected in this result to the effect that the transitions representing both working states are not contained. That means for the CPN that there is a marking from which there exists no path containing these transitions. In other words—from a COLA point of view—it is possible to reach a system configuration that prohibits a change to a distinguished system state (working in our case). Based on this information, the developer has to check whether the modelled system behaviour is what was desired. If this it not the case, a modelling error has been detected.

### 5.4.5. Related Work

Coloured Petri nets have been extensively used to model and verify business processes. Gottschalk et al. [82] translated Protos models, i. e., a popular tool for business process modelling, into Coloured Petri nets for simulation, testing, and

configuration reasons. Moreover, in the field of Web services, CPNs are used. There, questions concerning correctness and reliability arise, when composing single Web services to more complex ones. Kang et at. [111] and Yang et al. [201] have studied the translation of WS-BPEL (*Web Services Business Process Execution Language*) or BPEL specifications into CPNs. This makes analysis and verification of the composed Web services using for example CPN Tools [50] possible. However their translations are rather informal than a formal defined translation schema. Hinz et al. [97] translated BPEL specifications into Petri nets in order to use the model checking tool LoLA to verify relevant properties.

Akin to the presented approach, the authors of [74] bring together the two modelling languages UML and CPN. They translate Use Cases and UML 2.0 Sequence Diagrams into CPN models for formal analysis. In the automotive domain or in the field of embedded systems design in general, *Live Sequence Charts (LSC)* are widely used as specification language. Amorim et al. [13] claim that LSC do not provide the possibility for analysis and verification and thus a translation into CPN is appropriate and which is given in a well-defined formal way.

### 5.4.6. Summary

This section exemplary outlined the way of using model-to-model transformation—in the current achievement from COLA core to Hierarchical Coloured Petri nets—to quickly adopt the analysis capabilities of other tools into the COLA-IDE. This becomes possible since, both modelling formalisms are based on a well-defined formal basis, which allows a stepwise transformation of COLA's modelling concepts.

**input** : COLA automaton $\mathsf{AUT} = \langle n, \sigma, I \rangle, I = \langle Q, q_0, \Delta \rangle$,
$\Delta \subseteq Q \times \mathsf{dom}(\mathsf{in}(\sigma)) \times Q$

**output**: Hierarchical CPN $Hcpn = (S, SN, SA, PN, PT, FS, FT, PP)$

```
/* STEP 1                                                    */
```

1 $S = \{CPNautomaton\} \cup S_Q$

2 $SN = \{n_{\mathsf{AUT}}\} \cup SN_Q, n_{\mathsf{AUT}}$ is the identifier of AUT

3 $SA(SN) = \bigcup_{s \in SN} SA(s)$

4 $PN = \pi(\mathsf{io}(\sigma)) \cup PN_Q$

5 $PT(p) = \begin{cases} i/o & \text{if } p \in \pi(\mathsf{io}(\sigma)) \\ PT(PN_Q) & \text{if } p \in (PN_Q) \end{cases}$

6 $PA(t) = \begin{cases} \{(in_1@CPNautomaton, in_1@automaton), \\ (in_2@CPNautomaton, in_2@automaton), \ldots, \\ (out_1@CPNautomaton, out_1@automaton), \ldots\} & \text{if } t = n_{\mathsf{AUT}} \\ PA(SN_Q) & \text{if } t \in (SN_Q) \end{cases}$

7 $PP = 1`CPNautomaton$

8 *automaton* represents the subpage of $n_{\mathsf{AUT}}$.

```
/* Note:  FS and FT are not considered during the
   translation.                                              */
```

```
/* STEP 2                                                    */
```

9 $P = \{State, activated\} \cup \pi(\mathsf{in}(\sigma) \cup \mathsf{out}(\sigma))$

10 $T = \{activate\ State\} \cup Q_T$

11 $A = \{(State, activate\ State), (activate\ State, State),$
$(activate\ State, activated), (activated, activate\ State), \ldots\}$

12 $\Sigma = \{State, State_L, \} \cup \mathsf{D}, \text{ with } \mathsf{D} = \mathsf{dom}(\mathsf{in}(\sigma))$

13 $V = \{s : State\} \cup \{v_1 : t_1, \ldots, v_n : t_n\}, t_i \in \mathsf{D}, 1 \leq i \leq n, n = |\mathsf{in}(\sigma)|$

14 $C(p) = \begin{cases} State & \text{if } p = State \\ State_L & \text{if } p = activated \\ \mathsf{D} & \text{if } p \in \pi(\mathsf{io}(\sigma)) \end{cases}$

15 $G(t) = \textbf{true}, \forall\ t\ \in\ T$

16 $E(a) = \begin{cases} s & \text{if } a = (State, activate\ State) \\ state(s, \{v_1, v_2, \ldots\}) & \text{if } a = (activate\ State, State) \\ [state(s, \{v_1, v_2, \ldots\})] & \text{if } a = (activate\ State, activated) \\ \ldots \end{cases}$

17 $I(State) = q_0$

**Algorithm 4:** Translates a COLA automaton into a corresponding hierarchical CPN according to the given schema.

```
   input  : COLA model
   output : CPN model
1  while (not all units u ∈ U have been visited) do
2      perform a DFS traversal on the COLA model
3      switch (u instanceof) do
4          case (functional block)
5              if (u isA constant) then
6                  create a single place p
7                  initialise p accordingly
8              else
9                  FunctionalBlock(u)
10         case (network)
11             Network(u)
12             create a transition input to collect the incoming data
13             connect input according to the connections in u (channels)
14         case (automaton)
15             Automaton(u)
16         case (delay)
17             Delay(u)
18             initialise the translation
19 glue input and output places together, according to their connectivity in the
   COLA model
20 assign ports to sockets
```

**Algorithm 5:** COLA2CPN translation algorithm.

CHAPTER

# SIX

## Generation of Requirements Specification Documents

Requirements specification documents do not only define the functionality and constraints of a system (software, hardware, or both) to be developed, but are also part of the contract between an OEM and a supplier. High-quality source material is essential for the quality and reliability of the product to be developed. Inconsistency between different documents may also influence the meeting of deadlines within the development process and therefore have economic consequences. The following section 6.1 gives a short introduction, before the generated document structure is discussed in Section 6.2. Section 6.3 focuses on the technical realisation, which in turn facilitates the realisation of a *semantically tangible requirements document* presented in Section 6.4. Section 6.5 refers to the 'IEEE Recommended Practice for Software Requirements Specifications'. Finally, Section 6.6 sums up this chapter.

### Contents


**6.1. Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **120**

**6.2. Document Structure** . . . . . . . . . . . . . . . . . . . . . . . . . . . **121**

**6.3. Realisation** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **126**

**6.4. Semantically Tangible Requirements Documents** . . . . . . . . **129**

**6.5. Integration into the Context of IEEE Std 830-1998** . . . . . . . . . **131**

**6.6. Summary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **132**

## 6.1. Introduction

Requirements specification documents should contain all necessary information such that the reader of the documents understands what s/he has to realise. Therefore, clarity and comprehensibility are only two aims. Actually, industry partners demanded a better readability and also the use of formal languages to describe components if this supports understandability.

Oftentimes, one differentiates between different kind of requirements specification documents according to their respective purpose.

(i) A *functional requirements specification document* describes in detail the intended function from a solution independent point-of-view.

(ii) A *component requirements specification document* is aimed at the definition of a component, whereas the notion component is used in the sense of a hardware component—in general an ECU. This document specifies an ECU; note: several (sub-)functions can be partitioned onto a single ECU, thus, aspects of different functions can be contained.

(iii) A *system requirements specification document* contains information from a system's point-of-view and therefore contains information how to integrate hardware components together to realise the whole system.

The purpose of such a document cannot be characterised only in terms of the contained information, but also in terms of its reader: in most cases the functional requirements specification document is used as an in-house document for the function developers and function specialists. Sometimes also the suppliers is handed-over this document in addition to the main document, which summarises the component requirements. Actually, this is the most important document for the supplier as it contains information about all functions that have to be realised by a particular E/E component. Finally, the system requirements document is necessary for those who have to integrate the component within the overall system or automotive in this case.

Today, these documents are either hand-written, fully or partially generated using tools like DOORS. As authors are usually different persons, consistency between different documents has to be guaranteed, which is difficult, time-consuming and—of course—error prone. Usually dozens to hundreds of documents are referenced and also modified over their life time. This makes consistency

within all documents a very challenging task, which today is in many cases not possible.

Actually, it is very difficult to find a good trade-off between preciseness and incomplete specification. Immediately, it arises the question why a requirements specification document should be incomplete or imprecise? This has amongst others the following two reasons:

(i) Requirements specification documents involve over their life time. Especially in early versions, not all information is available. Only in rare cases this is rooted in poor requirements engineering. When realising the time line of automotive development, one notices that even not all technological possibilities that may be available five to seven years in the future—especially in the E/E area—are known. This underspecification leaves a margin for short-dated adaptions. A second reason for short-dated feature introductions is in response to competitors' innovations.

(ii) With requirements specification documents, OEMs define the scope of *what* to develop but seldom *how* to realise it. Then it is the task of the supplier to develop the desired functionality rather than to one-to-one realise what was written. Sure, the environment has to be stated by OEMs, i. e., the interface, bus connections, timings, etc. Within this context, suppliers have quite a lot of freedom.

Besides impreciseness, the contrary extreme is complete information. Complete ASCET-SD or MATLAB/Simulink models are provided for one-to-one realisation as appendix to the requirements specification documents. Hence, they become part of the contract between OEM and supplier.

In practice, however, terms like *inconsistency*, *incompleteness*, and even *incorrectness* are oftentimes attributed to requirements specification documents. In many cases this is because of inadequateness of natural language. Natural language is *imprecise*, *ambiguous*, and *confusing* when having realistic requirements documents with hundreds of pages in mind. As discussed above, incompleteness might be a desired feature, but then it has to be made clear that at a certain point this is made by intention.

## 6.2. Document Structure

The COLA automotive approach supports the generation of requirements specification documents. Through the whole document, a uniform structure is kept.

In the following, the proposed structure is given. One of the main advantages is that requirements are located directly next to the object they relate to. This approach guarantees that hundreds of requirements for instance for a single function are not spread all over the document, but are placed next to the respective functional description. This locality improves the readability and therefore the understandability of the function to be realised. The structure is leant against the Logical Architecture's structure. Assume the model on the Logical Architecture is structured as depicted in Figure 6.1. The illustrated figure gives only a partial

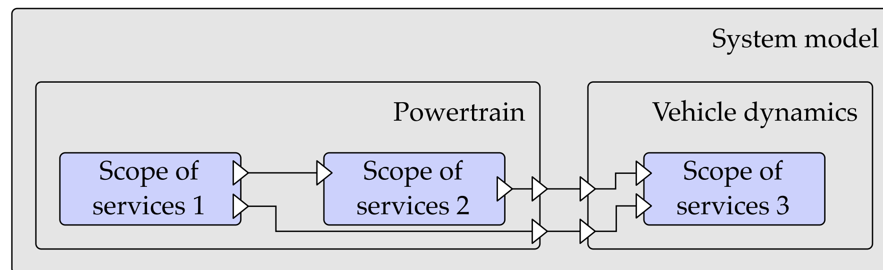



**Figure 6.1.:** Scope of services: the functional range to be part of the generated specification document.

excerpt of the overall system model. This functional model is structured according to some domains like for example *vehicle dynamics* or *powertrain*. Each of these domains in turn can be sub-divided. In this example in so-called *scopes of services*. Each of them encapsulates exactly that part of the overall system, a supplier shall develop and is responsible for. Therefore, in the granularity of the example, it is the scope of services, which is basis for the document to be generated. Everything contained in it will be part of the generated requirements specification document. This contains all requirements attached to units representing the scope of services, its sub-structure, a figure illustrating the logical behaviour, interfaces, and also the textual syntax of the respective COLA units, if desired. The textual syntax, which can be seen as a platform-independent pseudo code, eases readability and comprehensibility of the document.

### 6.2.1. Differentiation between Customer- and System Requirements

COLA units as modelling artefacts on the Logical Architecture have two possible reasons for their being. First, they are reused from the Feature Architecture and thus are referred to as *customer requirements*, as they have been modelled to

realise a customer feature. The second reason has a technical nature. Assume for example a logical signal *speed* on the Feature Architecture. Possibly, there may not be an isomorphic signal on the Logical Architecture and hence has to be computed first. The computation can be established using for instance the wheel rotation speed ('rotation speed' in Figure 6.2), wheel radius, and elapsed time, all available as sources, which are the logical representations of sensors on the Technical Architecture. Again, these additional computational efforts and modelling artefacts (X' and 'rotation speed' in Figure 6.2) are not directly rooted in the feature itself, but in its technical realisation. Actually the only modelling artefact directly derived from the feature is X. To have a clear differentiation

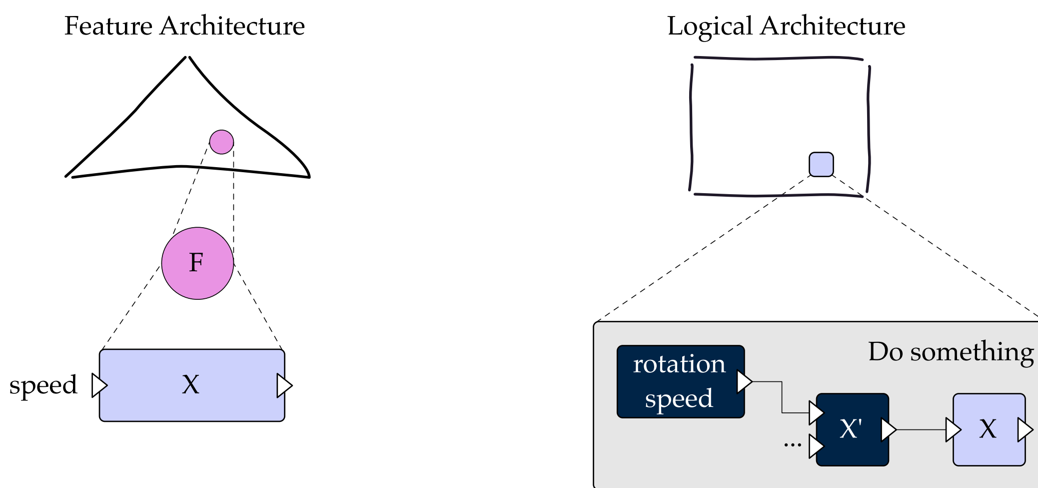



**Figure 6.2.:** Left: some feature F realised by a unit X.
Right: post-processing of source values. Dark shaded units are referred to as *system requirements*: 'rotation speed' and X'. X is a *customer requirement*.

between the two mentioned classes, they are placed and treated separately in the generated requirements specification document.

## 6.2.2. Structuring

In the following, the proposed structure of the generated document is given. Note, the document is generated from the model of the Logical Architecture and has a recursive structure.

To alleviate understanding of and navigation through the document, a generic structure is proposed, which is capable to be used for all three mentioned document types, namely the functional, the component, and the system requirements

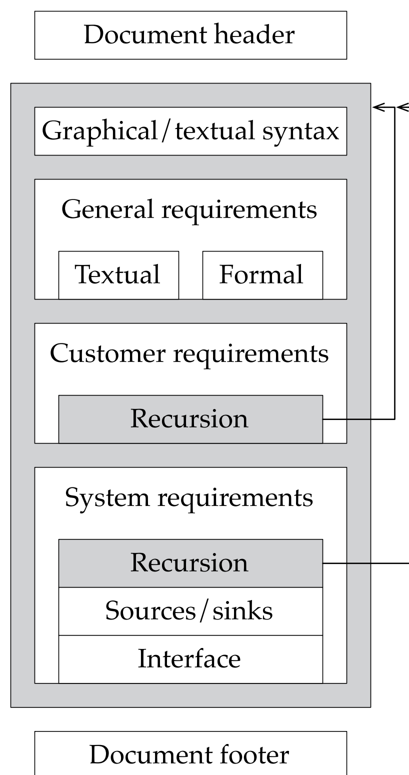



**Figure 6.3.:** Generic document structure.

specification document. Within this thesis, only the first mentioned will be discussed. However, at our industrial collaborator, also the generation of the component requirements specification document has been advised and supported right up to the integration into their tool chain for productive use.

Each document begins with a prosaic description of the function to be realised, followed by an image of that COLA unit encapsulating the scope of services to be realised, and its textual syntax. Note, this and other information may be subject to intellectual property (IP) of the OEM. As a consequence, only the interface and the requirements for that particular object are given. A unit marked as IP (`isIP()` is used in descend to obtain this information) is considered as a black box, whereas non-IP units are recursively descended and can therefore be considered as glass boxes.

Next, the requirements directly annotated or linked to a unit are listed. These are informal textual as well as formal requirements. The main advantage to place them next to the image is locality, i. e., it is always clear which requirements belong to which modelling artefacts and are not spread all over the document. The two main parts namely the

(i) customer requirements and the

(ii) system requirements

follow. Again, the IP issue becomes important at this point. The root unit is taken and for all contained sub-units that are marked as customer requirements, their graphical and textual representation, the annotated requirements, and their customer and system requirements are given, respectively. Subsequently, the system requirements are listed. Potentially contained sources and sinks, as well as the signature of the unit at hand is included. The generic layout of the generated requirements specification documents is depicted in Figure 6.3. Algorithm 6 shows the algorithm to generate such a document for a unit on the Logical Architecture.

In addition to the already contained information, in many cases it might be beneficial to know to what extend, the realised function is related to other components, i. e., it's *context* is of interest. Figure 6.4 depicts the context of the scope of services of the modelled example. The relation (expecting or providing information) to other domains, hardware entities, or other scopes of services is depicted as well.

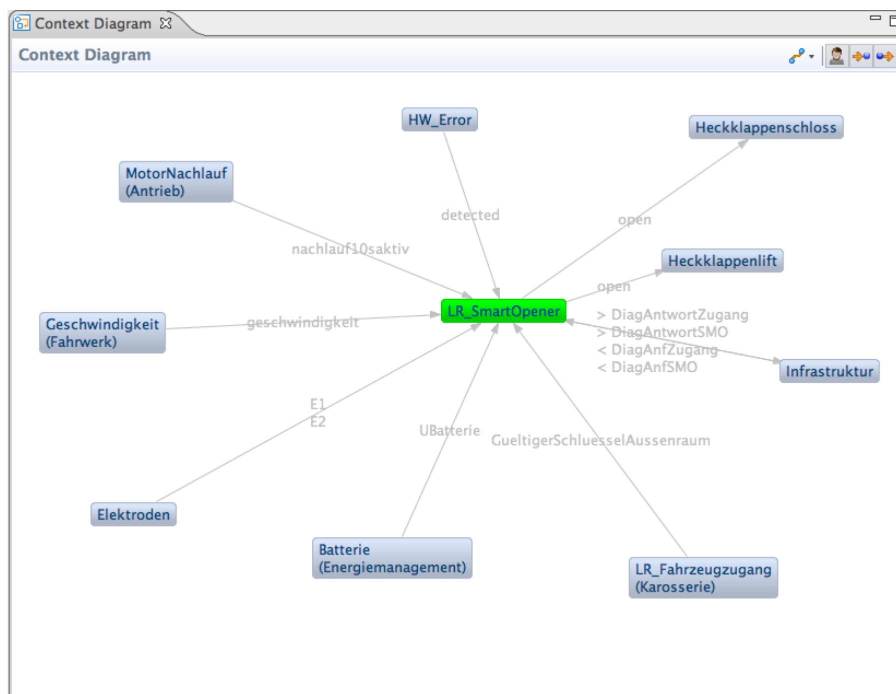



**Figure 6.4.:** Context of the scope of services of 'LR_SmartOpener'.

## 6.3. Realisation

Algorithm 6 and procedure descend, respectively, perform a depth-first-search (DFS) on the COLA model of the Logical Architecture. The user specifies a root unit $r$ where the model traversal is started (that encapsulating the scope of services). The document is generated analogously to the proposed generic structure respecting intellectual property. As the approach is generic, each non-atomic unit can be used therefore. However, in order to maintain a consistent level of abstraction, it is recommendable to use similar levels for different documents. In Figure 6.1 scopes of services have been introduced. These 'container' units encapsulate modelling artefacts and thus define the scope of the generated document.

```
   input  : Root unit r to start with
   output : Document structure
1  Generate document header
2  if (¬isIP(r)) then
3      Print graphical syntax
4      Print textual syntax
5  descend(r)
6  Generate document footer
```

**Algorithm 6:** Generates a requirements specification document.

When traversing the COLA model, two things are generated simultaneously and depicted in Figure 6.6:

1. An XML file, which is used to render the final document (cf. Figure 6.6b). The generation process is depicted in Figure 6.5.

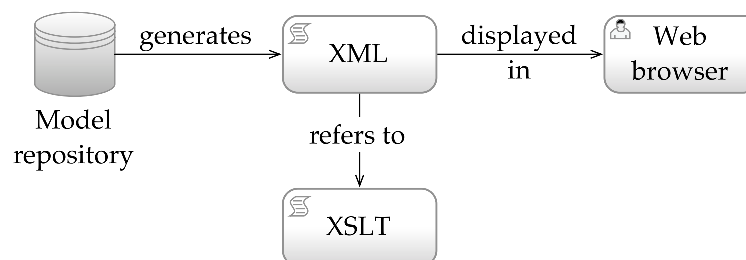
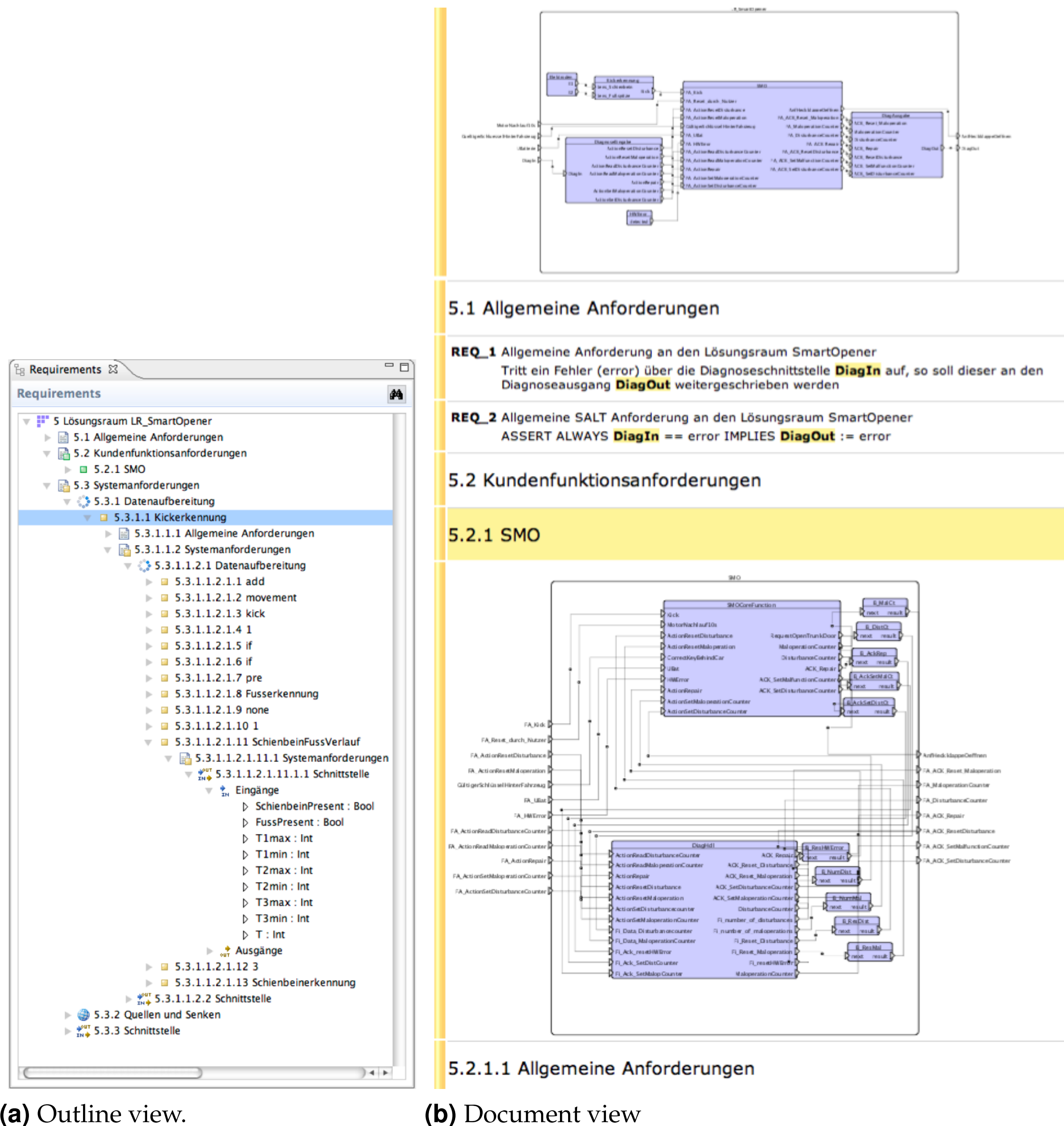



**Figure 6.5.:** Specification document generation and visualisation process.

```
1  Stack<Unit> stack
2  if (u has textual requirements) then
3      Print textual requirements
4  if (u has formal requirements) then
5      Print formal requirements
   /* Considering customer requirements                                   */
6  C ← sub-units of u that are marked as customer requirements
7  if (|C| > 0 ∧ ¬isIP(u)) then
8      stack.addAll(C)
9      while (stack.size() > 0) do
10         s ← stack.pop()
11         if (¬isIP(s)) then
12             Print graphical syntax
13             Print textual syntax
14         descend(s)
   /* Considering different system requirements
      manifestations                                                      */
15 D ← sub-units of u that are used for data processing, i. e., no customer
   requirements, and no sources/sinks.
16 if (|D| > 0 ∧ ¬isIP(u)) then
17     stack.addAll(D)
18     while (stack.size() > 0) do
19         s ← stack.pop()
20         if (¬isIP(s)) then
21             Print graphical syntax
22             Print textual syntax
23         descend(s)
   /* List sources and sinks                                              */
24 Print all sub-units of u that are sources
25 Print all sub-units of u that are sinks
   /* List the interface of u                                             */
26 Print all input ports of u
27 Print all output ports of u
```

**Procedure** descend(Unit *u*) performs a DFS.

2. An outline view, which is used within the COLA modelling environment to navigate through the generated document, is depicted in Figure 6.6a.




**(a)** Outline view.

**(b)** Document view

**Figure 6.6.:** Figure (a) shows the layout view of the generated document, and (b) gives a short extract of the document.

*Flexibility* is the main advantage of generating XML documents. The look-and-feel of the final document can easily be adapted by changing the XSLT file. This XSL Transformation file is used to transform the generated XML file into the

target structure. Hence, the constraints defined by an OEM with respect to layout, structuring, and corporate design can easily be met. Different target file formats like XHTML, PDF, or Excel are supported in principle.

The possibility to exclude marked units of interest from the generation process facilitates the protection of intellectual property of an OEM. Only those information of a marked unit relevant for the supplier are given: annotated requirements and the interface. The behavioural model of such a unit is not handed over to the supplier.

## 6.4. Semantically Tangible Requirements Documents

In the daily work of many automotive companies, requirements management is done using some commercial off-the-shelf tools. A predominant example is IBMs Rational DOORS, which is used to manage requirements and to establish links between them, yielding traceability. Using generators, requirements specification documents can be generated as Word or PDF documents. However, automatic document generation is not the rule. Usually, in an early phase of product development, a multitude of different product specification documents are manually written, or partially generated using requirements management systems (cf. [31]), as mentioned. It is a challenging task to maintain consistency between all those documents. Changes of one document may cause other documents to be modified, as well. Gained experience at our industry partners taught us the rule that requirements are only allowed to be written once in a document, or a collection of documents. Again, this is very hard to ensure, especially in hand-written and -maintained documents. IEEE Std 830-1998 [101] Section 4.3.7 states that redundancy is not an error, but can easily lead to errors and hence should be avoided.

To overcome these difficulties, this thesis proposes to generate all documents from a single data source—the central model repository—, which together with the sophisticated COLA engineering environment enables a rigorous specification document generation. Consistency amongst different generated documents can be guaranteed as long as they have been generated at the same time, i. e., they are based on the same model revision.

Through the tight integration of the requirements specification document generator into the COLA modelling environment a plenitude of key benefits have been reached:

(i) A single tool without the need to use error-prone adapters.

(ii) A single model repository storing *all* necessary artefacts, i. e., no possible inconsistent databases that have to be queried and tried to keep consistent.

(iii) The generated document can be seen as a special view or perspective on the modelled information. Thus, it does not make any difference, whether to directly edit the model in the *modelling perspective*, or by editing the generated document within the *specification document perspective*. Hence, it is possible to directly edit for example captions, attributes, etc. within the generated document. As a consequent next step, one can think of adding requirements to units, creating units or other modelling artefacts similar to a text processing tool, but in contrast with the full power of a well-defined semantics, clear syntax, and the possibility to perform consistency checks in the background, akin to the modelling perspective.

(iv) Moreover, in combination with the COLA model simulator [96] within the *simulation perspective*, the generated document becomes dynamic, i. e., by clicking highlighted elements, the corresponding modelling artefacts are shown. This synchronises the current position within the document with the behavioural model. Requirements annotated to modelling artefacts get a context, which is shown as the graphical COLA syntax. That makes it possible to click for example on signal names inside the document, and the respective ports are shown together with their units. It is even possible to run the simulator on a particular unit, selected in the document. That accounts for a much better understanding of the generated requirements specification document, because in addition to a list of requirements in the correct context, the desired behaviour can be simulated, assumed a behavioural model has been developed and the unit in question is not marked as IP. Furthermore, also test cases are given as requirements that have to be tested against. Thus, they are also present within the document, which in turn makes them executable by the simulator.

All those mentioned benefits are only possible because of

(i) the tight integration into the COLA modelling environment,

(ii) the generation based on a single data source (model repository),

(iii) the rigorous syntax and semantics of COLA, and finally

(iv) the coupling with the model simulator.

Figure 6.7 depicts a screen shot of the COLA-IDE making requirements specification documents semantically tangible. Basically, the IDE consists of four main parts namely the model viewer and its property editor, an outline of the generated document, the generated document itself, and on the right hand side the COLA simulator, which shows graphically the simulation steps specified in a test case, shown in the document view.

## 6.5. Integration into the Context of IEEE Std 830-1998

According to IEEE Std 830-1998 ('IEEE Recommended Practice for Software Requirements Specifications') [101], a software requirement specification should be *correct*, *unambiguous*, *complete*, *consistent*, *ranked for importance and/or stability*, *verifiable*, *modifiable*, and *traceable*.

These attributes and their compliance aim among other things at reducing the development effort and providing a baseline to organise the product's verification plan. In the following, arguments are given why the presented approach and in particular the created generate has the same characteristics as demanded above.

(i) **Correctness.** As [101] emphasises, no tool can guarantee correctness. In this context, correctness means that every requirement of the document is one the software shall meet. But assuming the requirements given by the user are correct, so the requirements put into the generated document are. The user should convince herself of the correctness before generating the document.

(ii) **Unambiguity.** As long as textual requirements are given by humans in an informal way, ambiguity cannot be excluded in any case. But the COLA automotive approach offers the possibility to formulate requirements as SALT specifications, which are unambiguous and intended to be processed electronically. Moreover, when regarding the included parts (textual or graphical) of the COLA model, ambiguity is out of question due to the well-defined syntactical and semantical COLA definition.

(iii) **Completeness.** Similar to correctness, completeness is difficult to evaluate. The standard gives examples for significant requirements such as for instance those relating to the functionality, performance, design constraints, or external interfaces. Of course, references to tables, figures, etc. as well

as definitions and unit measures shall be given. As far as the user provides these information, they are generated, too.

(iv) **Consistency.** A requirements specification is said to be 'consistent if, and only if, no subset of individual requirements described in it conflict' [101]. For requirements given as SALT specifications, one can effectively check whether they conflict as outlined in Section 5.2. Of course, this might also be possible when using other formal requirements specification formalisms and techniques.

(v) **Ranking for importance and/or stability.** COLA's requirement objects can be attributed with importance and /or stability weights.

(vi) **Verifiability.** A requirement is verifiable according to [101] 'if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirements'. Obviously, this is true for SALT specifications. For prosaic requirements, it is up to the requirements engineer to be precise and keep in mind that there has to be an actually executable test.

(vii) **Modifiability.** IEEE Std 830-1998 recommends that changing the requirements should retain the structure and the style of the specification document while being easy to make and staying complete and consistent. Since the document is generated in accordance with a predefined structure, modifiability is given.

(viii) **Traceability.** The relevant standard demands both forward and backward traceability of requirements. To accomplish this demand, the COLA-IDE and the COLA automotive approach in general have to be extended as outlined in the outlook (cf. Section 9.2). Currently, only the attachment of a requirement object to each COLA entity is supported, but no life-cycle management and dedicated requirements management. If this was the case, the generated document could easily be extended with respect to the desired traceability capabilities.

## 6.6. Summary

This chapter outlined COLA's capabilities to automatically generate a requirements specification document directly from the model on level of the Logical

Architecture. It was shown how one type of document can be generated. Of course, this approach can be extended to generate different documents for different stakeholders. When all those documents are generated simultaneously, it can be guaranteed that their particular content is consistent. The generator was integrated into the COLA-IDE using its plugin mechanism, which allows a very tight coupling to other plugins. In particular, the coupling to the COLA simulator makes requirements documents come alive. By relating requirements coming along with their descriptions with the behavioural model, their meaning can be explored and verified by running the simulator.

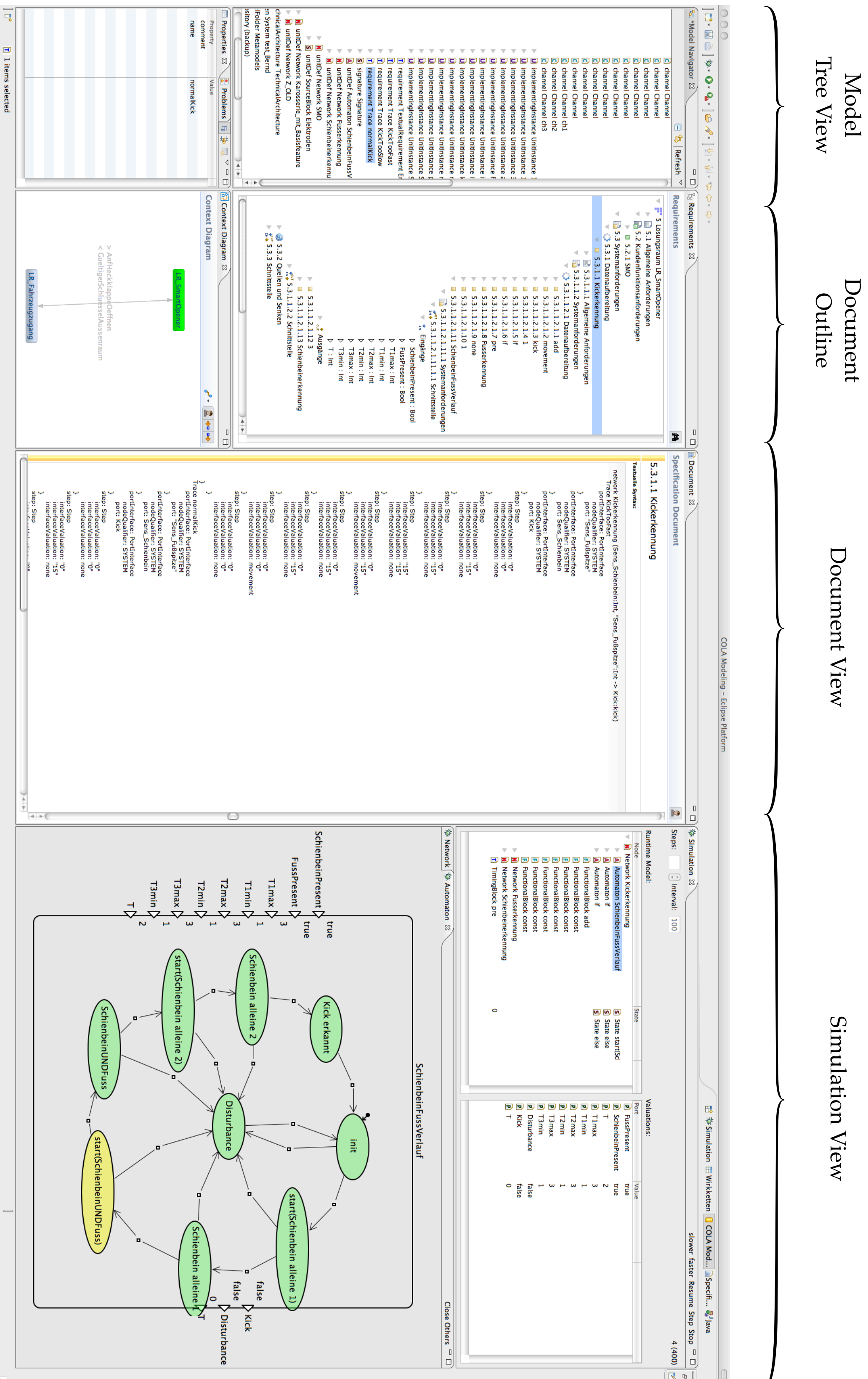



**Figure 6.7.:** Semantically tangible requirements specification document.

CHAPTER
# SEVEN

# Deployment

The platform independent model (model on the Logical Architecture) together with the platform dependent model (model on the Technical Architecture) provide enough information to deploy a logically described functional model (behavioural model) onto the target platform. Therefore, a couple of steps are necessary and briefly described in Section 7.1. This chapter focuses on aspects that relate to software onto hardware allocation in Section 7.2 and the timely execution on the target platform in Section 7.3. These two techniques are in a next step extended to realise fault tolerant systems based on the COLA paradigm. Details are given in Section 7.4, followed by related work and a summary in Sections 7.5 and 7.6.

**Contents**


7.1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
7.2. Allocation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
7.3. Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
7.4. Fault Tolerance . . . . . . . . . . . . . . . . . . . . . . . . . . 169
7.5. Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
7.6. Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179


## 7.1. Introduction

One of the key ideas behind the COLA automotive approach is to illustrate the feasibility of a seamless model-based development of safety-critical embedded systems in the automotive domain. It is strongly believed that this approach can

only work, when manual user manipulation of produced artefacts is reduced to a minimum. This prevents the accidental insertion of errors to the greatest possible extent. On the one hand, this does not exclude user interactions with the system for example when formal techniques like model checking were applied and the user has to improve the model according to possible errors. On the other hand, the user should not be allowed to modify generated code, just to mention one example.

Therefore, the COLA approach provides a fully automatic deployment concept. Assuming the deployment steps are free of errors and the model of the Logical Architecture indeed does what has been demanded on the Feature Architecture, a correct behaviour on the target platform can be guaranteed *by construction*. Of course, hardware failures can only be tolerated to a certain extend, which will be discussed in Section 7.4. This approach follows the idea of a *systems compiler*, which—similar to a C compiler, for example—builds from a specification an executable object. Classically, programs are written in some kind of high-level programming language, which is then transformed into assembly and machine code. Hereby optimisations are considered. A systems compiler in turn automatically builds a runnable entity by transformation from a behavioural model. The transition from a feature model to a functional model is generally a step involving user input. However, tool support allowing only distinct operations on models, for instance merge (cf. [38]) or split, inhibits user-inserted errors. Non-functional requirements are objectives for optimisation. From an industrial point of view, automotive software as well as hardware architectures oftentimes follow different criteria: the organisational structure of a company plays in many cases an important role. However, the idea of a systems compiler and a push-button deployment as we outlined in [86,87] sets the agenda. This in turn is confirmed by Sangiovanni-Vincentelli and Di Natale [177] who state that the optimal approach

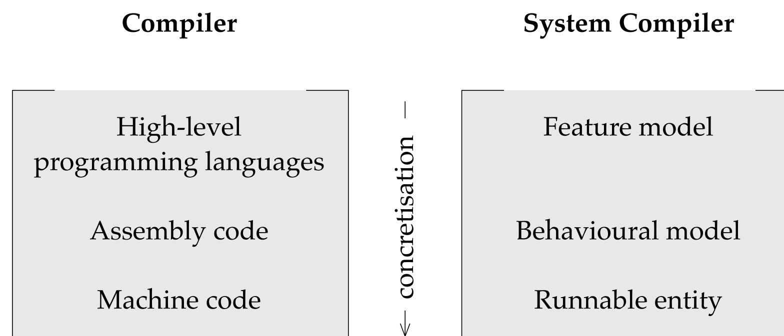



**Figure 7.1.:** Systems compiler.

would automatically allocate tasks to computing nodes guaranteeing a correct system behaviour, and using the resources optimal. The presented way of doing deployment can be considered to be static, i. e., it is performed at design time.

Before proceeding with the preliminaries, it is useful to recapitulate the steps performed during deployment: **(S0)** *Modelling and Verification*, **(S1)** *Partitioning*, **(S2)** *C Code Generation*, **(S3)** *Resource Estimation*, **(S4) Allocation**, **(S5) Scheduling**, and **(S6)** *Configuration*. This thesis focuses on the highlighted steps **(S4)** and **(S5)**.

### Preliminaries

The COLA automotive approach cannot be stated to be all-purpose applicable. That does not imply any restrictions, but defines a setting and methodology of modelling safety relevant automotive systems. The presented approach abandons by intention aspects like event-triggered handling, dynamic scheduling, and manual code manipulation. This is done in consideration of one main goal: *predictability*. The run-time behaviour is completely predictable in COLA-powered systems. Figure 2.1 depicts the common setup of a controlled automotive system. To reflect COLA's semantics as close as possible and thus allow semantics preservation from feature modelling down to the deployed system, COLA systems are cyclically executed as shown in Figure 7.2. The evaluation of COLA systems

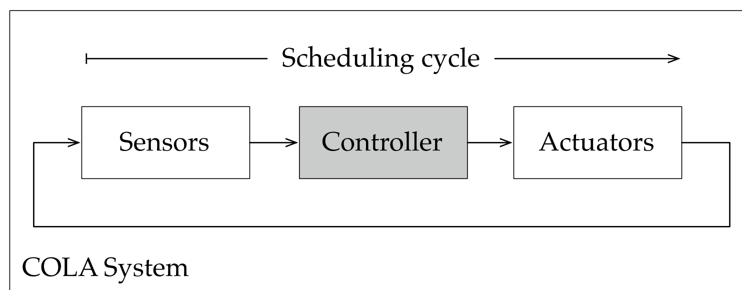



**Figure 7.2.:** Scheduling cycle.

follow the three phases according to the IPO-Model, i. e., Input-Process-Output Model:

**Input** Read *sensor* values.

**Process** Perform the modelled functionality in order to *control* the system and generate fresh output values.

**Output** Write the new values to the *actuator*s.

This model of execution fits best to the synchronous data-flow semantics of the COLA core modelling language with its periodic execution.

As mentioned, in favour of predictability, COLA and the COLA automotive development concept base on the time-triggered paradigm. Here, tasks are executed at specific, predefined points in time. This guarantees a timely and thus correct system execution.

**AUTOSAR**

In 2005, when the research project with BMW Group started, the introduction of *AUTOSAR* just began. At that time it was open whether AUTOSAR will be a success story, as the first specification had too many drawbacks (v.1.0). If, and only if, a supplier builds an AUTOSAR software component (SWC) for more that one OEM, AUTOSAR can gain leverage. In 2005, that was anything but sure. Hence we decided to develop our own middleware reported by Haberl et al. [85]. Today, the situation is different with the current version 4.0.3.

Similar to AUTOSAR's **V**irtual **F**unction **B**us (VFB) the developed middleware abstracts from the underlying network topology, thus communication and information exchange becomes transparent. Furthermore, the middleware is used to store system-wide information, in particular operating modes (automata states) and delay values, and synchronises the clocks of all involved ECUs. A schematic construction of the system architecture including the middleware is given in Figure 7.3. A **R**eal-**T**ime **O**perating **S**ystem (RTOS) is executed on the hardware

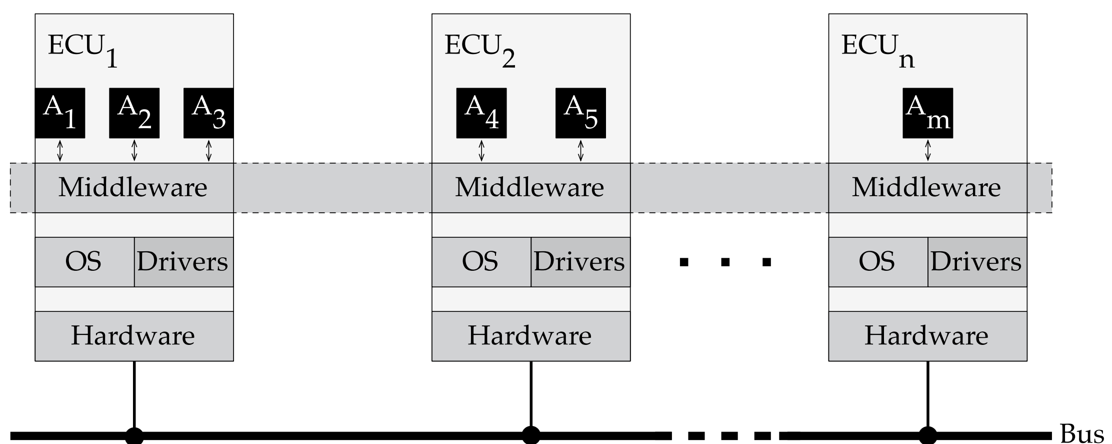



**Figure 7.3.:** System Architecture.

**real-time operating system**

platform. Applications ($A_1$ to $A_m$) communicate via a middleware running on each ECU ($ECU_1$ to $ECU_n$).

## 7.2. Allocation

As soon as the model of the Logical Architecture has been partitioned into deployable entities (clusters), allocation takes place. In this process step, an optimal placement of clusters onto ECUs (actually onto a processor as a part of an ECU) is determined. In the explanation below, only ECUs with a single processor are considered. As multi-core and multi-processor ECUs become more and more important, the presented approach can easily be extended towards multi-core architectures (cf. [129] for example).

Of course, the fulfilment of all functional requirements has to be ensured by the allocation step, however, allocation is rather uncritical in this respect. Later on the scheduling step has to ensure that the model's semantics is preserved, which in turn highly influences the functional—or better the *correct* functional—behaviour.

This section describes in detail, how **N**on-**F**unctional **R**equirements (NFR) are used to guide the allocation towards an optimal solution. For the non-functional requirements exemplified in Figure 7.5, their consideration is outlined below (Section 7.2.2). But before that, the used notation is explained in the following section.

**non-functional requirement**

### 7.2.1. Notation

Let $C = \{c_1, c_2, \ldots, c_n\}$ denote the set of clusters to be mapped onto the set of all processors (ECUs) $P = \{p_1, p_2, \ldots, p_m\}$ with $n, m \in \mathbb{N}$ and $n$ being the number of clusters and $m$ the number of processors, respectively. In the following, the indicator variable $a_{[c \mapsto p]}$ is used where $c \in C$ denotes the cluster to be mapped onto the processor $p \in P$. It indicates whether there is a mapping or not

$$a_{[c \mapsto p]} = \begin{cases} 1 & \text{if cluster } c \text{ is allocated onto processor } p \\ 0 & \text{otherwise.} \end{cases}$$

Hardware capabilities as well as software requirements have to be considered in order to optimise the placement. Therefore, attributes of both software (clusters) and hardware (ECUs) have to be taken into account. Table 7.1 summarises the attributes. Assume we have two relations $\mathscr{R}$ used to store information about the needed resources to execute clusters $c_i$ on processors $p_j$ with $1 \leq i \leq n$, $1 \leq j \leq m$

and $\mathscr{P}$ holding the capabilities of the processors.

| $\mathscr{R}$ | | | | | | |
|---|---|---|---|---|---|---|
| cluster | cpu_cycles$_{p_1}$ | ... | cpu_cycles$_{p_m}$ | rom_req | ram_req | ... |
| $c_1$ | * | ... | * | * | * | ... |
| $c_2$ | * | ... | * | * | * | ... |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $c_n$ | * | ... | * | * | * | ... |

In the data model, the relation $\mathscr{R}$ is realised as an association class (cf. Figure 7.4), yielding a separation of the architectural/hardware model and the resource model.

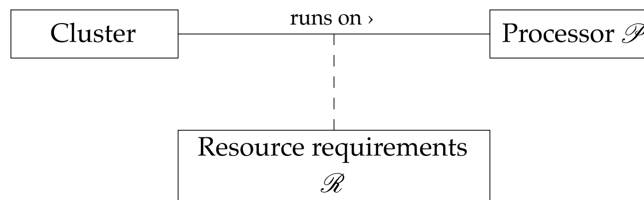



**Figure 7.4.:** Separation of architectural and resource model.

| $\mathscr{P}$ | | | | | | |
|---|---|---|---|---|---|---|
| processor | costs | rom_cap | ram_cap | os_overhead | power_state | ... |
| $p_1$ | * | ... | * | * | * | ... |
| $p_2$ | * | ... | * | * | * | ... |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $p_m$ | * | ... | * | * | * | ... |

Note, for the sake of clearness, not all attributes are listed. The complete compilation is given in Table 7.1.

We use the classical operators $\sigma$ (selection) and $\pi$ (projection) known from relational algebra to query for information stored in the relations $\mathscr{R}$ and $\mathscr{P}$. For instance if we are interested in the set of processors having a price of at most $c$, we write $\pi_{processor}\left(\sigma_{costs \leq c}\left(\mathscr{P}\right)\right)$, i. e., we first select those tuples satisfying $costs \leq c$ and then restrict the result to the attribute processor.

In the remainder of this thesis, the following notation is used to determine the placement of a cluster and converse the set of clusters mapped onto an ECU.

**Definition 8** (Mapping function)**.** *Let $\alpha : C \rightarrow P$ be the function mapping clusters $C$ onto ECUs $P$. Vice versa, $\alpha^{-1} : P \rightarrow 2^C$ determines the set of clusters allocated onto an ECU.*

| | Attribute | Description |
|---|---|---|
| Hardware attributes | costs | The accruing costs when using the ECU. |
| | rom_cap | The amount of non-volatile memory used for binary program storage. |
| | ram_cap | The amount of dynamic memory used during program execution. |
| | os_overhead | The amount of operating system overhead incurred when executing a cluster (dispatching, memory management, etc.). |
| | power_state | The lowest power state an ECU is active in. |
| | supplier | The supplier's name building the ECU. |
| | processor_arch | The processor architecture. General-purpose processors, DSP, etc. can be distinguished. |
| | proc_cycles | The available number of processor cycles available per millisecond. |
| Software requirements | cpu_cycles | The amount of processing cycles needed to execute cluster $c$ on processor $p$, denoted with $\pi_{cpu_cycles_p}(\sigma_{cluster=c}(\mathscr{R}))$. |
| | rom_req | The amount of non-volatile memory needed for binary file storage. |
| | ram_req | The amount of dynamic memory used during cluster execution. |
| | power_state | The name of the minimal power state the cluster can be executed. |
| | supplier | The name of the supplier implementing the cluster. |
| | replicas | The number of cluster copies distributed over the system for redundancy reasons. |

**Table 7.1.:** Attributes of clusters and ECUs used for the allocation.

### 7.2.2. Constraints

According to the classification of Figure 7.5, constraints are divided into *essential* and *auxiliary* non-functional requirements (NFR). An NFR is called *essential*, if it has to be satisfied to ensure a correct system operation. If for example an electronic control unit does not provide enough storage memory (ROM) for the cluster's binary file representation the cluster cannot be mapped onto the ECU.

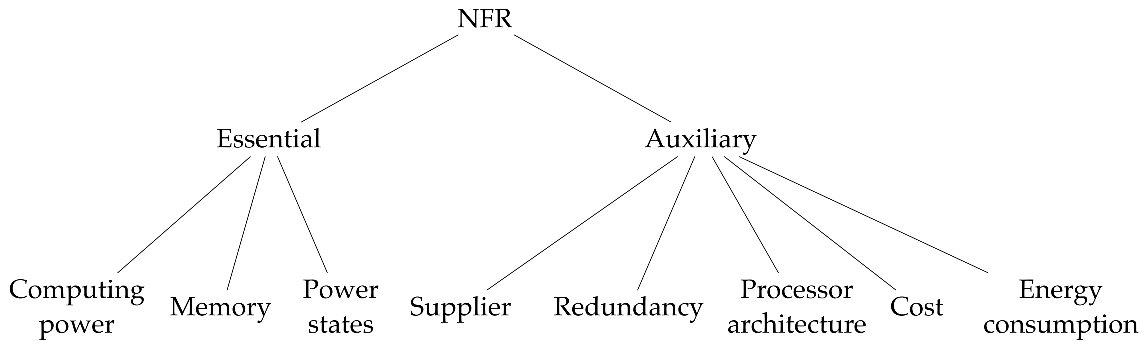



**Figure 7.5.:** Classification of non-functional requirements.

As a consequence, the violation of a single essential NFR may cause an unsafe or wrong operation of the modelled system.

The second category subsumes non-functional requirements that are not essential for a correct operation. Thus, they are referred to as *auxiliary* non-functional requirements. Their fulfilment, however, may improve the system in terms of quality attributes: allocation of different clusters from the same Tier 1 supplier onto a single ECU can improve the maintenance process, or a selective placement of clusters onto cheap or energy-saving ECUs can reduce costs and the $CO_2$ footprint, respectively.

### Essential NFRs

In the following list, the considered essential non-functional requirements are formalised. Beginning with the available processing power. The allocation of clusters onto the processors must not exceed the available resources, i. e., it holds $\forall p \in P$

$$\sum_{c \in C} a_{[c \mapsto p]} \cdot \left[ \pi_{cpu_cycles_p} \left( \sigma_{cluster=c} \left( \mathscr{R} \right) \right) + \pi_{os_overhead} \left( \sigma_{processor=p} \left( \mathscr{P} \right) \right) \right] \cdot \varrho \leq \pi_{proc_cycles} \left( \sigma_{processor=p} \left( \mathscr{P} \right) \right) \tag{7.1}$$

where $\varrho$ denotes the number of cluster invocations per time unit. As we will see later on, not all clusters allocated onto a processor are executed, but rather the current mode of operation determines the set of clusters responsible for rendering the modelled behaviour. The assumption, taken above guarantees a safe bound, even in the case of hardware failures discussed in Section 7.4. Next, the memory consumption is regarded. Here, one distinguishes between dynamic memory (RAM), and non-volatile memory (ROM) for storage of the binary code. It holds $\forall\, p \in P$

$$\sum_{c \in C} a_{[c \mapsto p]} \cdot \pi_{ram_req}\left(\sigma_{cluster=c}\left(\mathscr{R}\right)\right) \leq \pi_{ram_cap}\left(\sigma_{processor=p}\left(\mathscr{P}\right)\right) \tag{7.2}$$

and

$$\sum_{c \in C} a_{[c \mapsto p]} \cdot \pi_{rom_req}\left(\sigma_{cluster=c}\left(\mathscr{R}\right)\right) \leq \pi_{rom_cap}\left(\sigma_{processor=p}\left(\mathscr{P}\right)\right) \tag{7.3}$$

Besides the already mentioned essential NFRs, the power states ($ps$) a processor supports have to be considered. It holds $\forall\, ps \in \pi_{power_state}\left(\mathscr{P}\right)$ and $\forall\, c \in \pi_{cluster}\left(\sigma_{power_state=ps}\left(\mathscr{R}\right)\right)$

$$\sum_{p \in \pi_{processor}\left(\sigma_{power_state \geq ps}\left(\mathscr{P}\right)\right)} a_{[c \mapsto p]} = 1 \tag{7.4}$$

This ensures that a cluster with a specified power state is allocated onto exactly one processor with compatible power state.

### Auxiliary NFRs

Next, auxiliary non-functional requirements that do not change the functional behaviour but improve the system's quality of service are considered. In some cases it might be beneficial to allocate all clusters developed by a supplier $s \in \pi_{supplier}\left(\mathscr{R}\right)$ onto an electronic control unit also provided by him or her. It holds $\forall\, s \in \pi_{supplier}\left(\mathscr{R}\right)$ and $\forall\, c \in \pi_{cluster}\left(\sigma_{supplier=s}\left(\mathscr{R}\right)\right)$

$$\sum_{p \in \pi_{processor}\left(\sigma_{supplier=s}\left(\mathscr{P}\right)\right)} a_{[c \mapsto p]} = 1 \tag{7.5}$$

This ensures that a cluster with certain supplier demands is allocated onto exactly one processor from the same supplier. For redundancy reasons, a cluster $c$ can be allocated onto $r$ different, redundant processors, such that in case of for example a hardware failure, the cluster can be executed on a different processor. If no redundancy allocation is given, then $r = 1$, otherwise $r = \pi_{replicas}\left(\sigma_{cluster=c}\left(\mathscr{R}\right)\right)$. It holds $\forall\, c \in C$

$$\sum_{p \in P} a_{[c \mapsto p]} = r \tag{7.6}$$

The allocation decision is computed at design time and thus can be considered as *static deployment*. However, in case of the mentioned redundancy allocation, the ECU where a cluster is executed may change. More details are given in Section 7.4. For some clusters, it might be beneficial to be executed on a special processor architecture ($a$). Hence, annotations can be made accordingly. It holds $\forall\, a \in \pi_{processor_arch}\left(\mathscr{P}\right)$ and $\forall\, c \in \pi_{cluster}\left(\sigma_{processor_arch=a}\left(\mathscr{R}\right)\right)$

$$\sum_{p \in \pi_{processor}\left(\sigma_{processor_arch=a}\left(\mathscr{P}\right)\right)} a_{[c \mapsto p]} = 1 \tag{7.7}$$

In order to account for communication costs, a new indicator variable defined as follows is introduced:

$$c^{[c_i \mapsto p_u]}_{[c_j \mapsto p_v]} = \begin{cases} 1 & \text{if } a_{[c_i \mapsto p_u]} + a_{[c_j \mapsto p_v]} = 2 \quad (c_i \text{ and } c_j \text{ communicate}) \\ 0 & \text{otherwise.} \end{cases}$$

Inter- and intra-processor communication is important in a real-time system to consider communication delays and deadlines. If two clusters $c_i$ and $c_j$ are allocated onto the same ECU, communication for example over shared memory is done. This is very fast compared to communication over buses and gateways, necessary in an inter-processor communication scenario.

It holds for communicating clusters $c_i$ and $c_j$ that $c^{[c_i \mapsto p_u]}_{[c_j \mapsto p_v]} = 1$ if, and only if, $a_{[c_i \mapsto u_u]} = 1$ and $a_{[c_j \mapsto p_v]} = 1$, which formulated as linear constraints yields for all $1 \leq i, j \leq |C|$ and $1 \leq u, v \leq |P|$

$$-a_{[c_i \mapsto p_u]} - a_{[c_j \mapsto p_v]} + c^{[c_i \mapsto p_u]}_{[c_j \mapsto p_v]} > -2 \tag{7.8}$$

and

$$-2c^{[c_i \mapsto p_u]}_{[c_j \mapsto p_v]} + a_{[c_i \mapsto p_u]} + a_{[c_j \mapsto p_v]} = 0 \qquad (7.9)$$

These indicator variables are then multiplied by measured costs for inter- and intra-ECU communication. These costs include, amongst others, the communication frequency. Both, indicator variables and costs form the basis for a metric in the optimisation function.

In a similar fashion, different kinds of costs can be added. In the following economic costs are given for instance. Analogously, other constraints can be added. Hardware is an important expense factor. Hence, unused components like controllers, buses, and connection interfaces are only assembled if for example the costs for future extensions will be reduced. If a bus is exclusively used by unnecessary ECUs, i. e., no cluster is mapped onto them, it can be economised. This scenario, which is representative for similar dependencies, can be expressed as follows. It holds $\forall\, c \in C$ and $\forall\, p \in P$

$$-a_{[c \mapsto p]} + c_p > -1 \qquad (7.10)$$

and

$$-c_p + a_{[c \mapsto p]} = 0 \qquad (7.11)$$

where $c_p \in \{0, 1\}$ indicates that the expenses for ECU $p$ have to be taken into account during the optimisation process. Similarly, constraints for any other hardware component can be stated.

In some cases, it might be beneficial to allocate a cluster $c$ onto a fixed ECU $p$. In this case the simple constraint

$$a_{[c \mapsto p]} = 1 \qquad (7.12)$$

has to be added.

All the mentioned constraints, and other conceivable constraint extensions have to be fulfilled such that it is possible to find an (optimal) solution. Additional

requirements include for example maintainability, extensibility, and locality of input/output hardware. Maintainability demands for a placement of related clusters onto the same or a small number of ECUs. This results in fewer system nodes (ECUs) involved in software maintenance activities. Considering future functionality improvements, it may be beneficial to include some spare system capacity. This can be achieved by introducing dummy clusters. Regarding bus communication, it is convenient to place clusters involved in environmental interaction onto that ECU the respective sensors and actuators are connected to. To make optimisation possible, in the following an objective function is given.

#### Optimisation Function

Besides the given constraints, it is mandatory to define an optimisation function. It consists of the two main components *costs* and *metrics*. Costs characterise actual expenses whereas metrics subsume non-functional optimisation factors like memory, CPU time, or communication costs. For example the costs for ECUs ($costs_{proc}$) sum up to $costs_{proc} = \sum_{p \in P} c_p \kappa_p$ where $\kappa_p = \pi_{costs}\left(\sigma_{processor=p}\left(\mathscr{P}\right)\right)$ is the cost per unit obtained from the **B**ill **O**f **M**aterial (BOM). Metrics can be gained in a similar way. Generally, costs $costs_j$ and metrics $metric_k$ have to be minimised with respect to the constraints given in (in)equations (7.1) to (7.12), i. e.,

$$minimise \sum_j \lambda_j\, costs_j + \sum_k \mu_k\, metric_k. \tag{7.13}$$

The distinct but fixed weightings $\lambda_j$ and $\mu_k$ enable to characterise OEM's optimisation criteria. However, it might be challenging to determine these weighting factors, or at least requires experience.

### 7.2.3. Realisation

Formulae (7.1) to (7.12) define a linear system of (in-)equalities. Note, equalities in the mentioned formulae can easily be rewritten as two inequalities. Formula (7.13) defines the optimisation function. Together they describe a linear programming problem. As the optimisation function is linear and all coefficient of the case study were integers, the problem is a linear integer programming problem. Hence, a suitable solver (GLPK [2], the **G**NU **L**inear **P**rogramming **K**it) was used for the problem statement.

As mentioned above, the determination of the weights $\lambda$ and $\mu$ may be difficult. Therefore, one can also take into consideration to formulate the problem as a

multi-criteria or multi-objective optimisation problem. There, the aim is not to minimise a single objective, but multiple objectives simultaneously. That means, one could abandon the weights. In consequence, one can give for each cost and metric optimisation aim a separate objective function.

$$\underset{\mathbf{x}}{minimise} \left[cost_1(\mathbf{x}), \ldots, cost_j(\mathbf{x}), metric_1(\mathbf{x}), \ldots, metric_k(\mathbf{x})\right] \tag{7.14}$$
$$such\ that \text{ (in)equations (7.1) to (7.12) are satisfied}$$

where x is the vector of occurring decision variables.

The problem formulation as a multi-criteria optimisation problem would be the consequent next step since in many cases conflicting criteria occur.

The result contains assignments for all decision variables occurring in the linear programming or the multi-criteria optimisation problem. The assignments of the indicator variables $a_{[c \mapsto p]}$, for all $c \in C, p \in P$ determine the allocation or mapping function $\alpha$.

## 7.3. Scheduling

In order to fulfil a certain functionality, a hard real-time system has to execute a set of possibly concurrent tasks. Each such time-critical task has certain properties that the hard real-time system has to meet, such as its deadline. The underlying system has to guarantee that every task receives its required computational and data resources. The problem of finding a feasible allocation of there resources is called the *scheduling problem*.

### 7.3.1. Terminology

Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of $n$ clusters. Since the clusters are executed periodically, we observe an infinite sequence of cyclic cluster invocations, which are called *jobs* $c_{i,j}$ with $j \in \mathbb{N}^+$. We assume for each job $c_{i,j}$ to have the same characteristic as the corresponding cluster $c_i$. Each job is characterised by its *release time* $r_{i,j}$, *start time* $s_{i,j}$, *finishing time* $f_{i,j}$, *response time* $R_{i,j}$, i. e., $f_{i,j} - r_{i,j}$, *absolute deadline* $d_{i,j}$, and its worst-case execution time $C_{i,j}$. As we assume that each job has the same worst-case execution time, the same deadline, and the same release time, these values are referred to as $C_i$, $D_i$, and $R_i$, respectively. We denote with $\gamma$ the *cycle time* of the periodic execution. Figure 7.6 clarifies the notation.

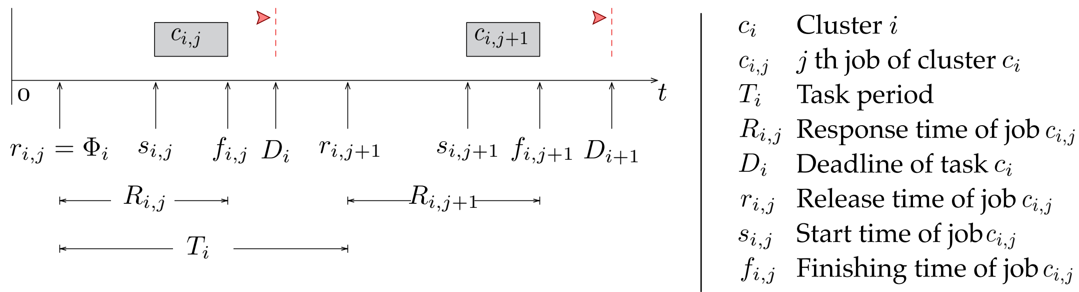



**Figure 7.6.:** Terminology for periodic scheduling.

### 7.3.2. A Taxonomy of Real-Time Scheduling Algorithms

Cheng et al. [46] present a taxonomy of real-time scheduling algorithms depicted in Figure 7.7. As the presented approach focuses on hard real-time systems, i. e.,

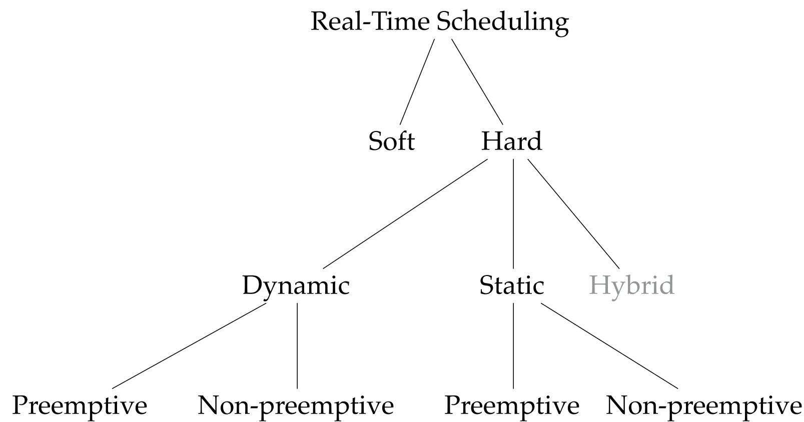



**Figure 7.7.:** Scheduling taxonomy referring to [46].

systems where one considers an operation as useless if its completion is after the specified deadline, i. e., too late, in the following only those systems are regarded. Similarly, if a result is delivered too early (for example an airbag inflates too early), it is considered incorrect.

One can distinguish between *static* (pre-run-time, or offline) and *dynamic* (or online) scheduling algorithms. The difference is the time *when* the algorithm is executed: static algorithms make their decisions at compile time, whereas dynamic approaches make their scheduling decisions at run-time.

In the following, both approaches are illuminated with respect to the mentioned attributes.

**dynamic scheduling**

A dynamic scheduler selects at run-time one of the currently ready tasks. Hence, a dynamic scheduling algorithm is very *flexible* as it is able to adopt with regard

to the current task set. The system can react on events that might not have been considered by the developer in such a way that occurring events immediately cause system activities. This flexibility, however, is bought dearly since managing semaphores, blocking mechanisms, etc. yield a high run-time overhead and prohibit—to a large extend—predictability, which is especially in the context of safety-critical hard real-time systems a desired property. Moreover, efforts for finding a feasible run-time schedule can be substantial. Usually, run-time scheduling decisions are made based on priorities and rules are applied, such as for example EDF (**E**arliest **D**eadline **F**irst). It is the most common dynamic priority-based algorithm for real-time systems. Priorities are assigned according to the deadline: a task receives the highest priority if its deadline is the earliest amongst the set of all ready tasks. Besides periodic task invocations, this algorithm is also well suited for aperiodic tasks. In 1973, Liu and Layland [140] provided a *necessary* and *sufficient* condition to check whether a set $C = \{c_1, c_2, \ldots, c_n\}$ of $n$ periodic clusters (tasks) is schedulable or not. They showed that the task set is schedulable by EDF if and only if

**earliest deadline first**

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{7.15}$$

holds. In fact, EDF is *optimal* amongst all dynamic scheduling algorithms, i. e., if a given task set is not schedulable by EDF, then it cannot be scheduled by any other algorithm [56]. Optimal means that it will always find a schedule, if an *exact* schedulability test states the existence of such a schedule. According to Kopetz [117], an exact schedulability test will always determine for a set of ready task whether they can be scheduled such that all tasks meet their deadlines. *Sufficient* schedulability tests can be easier as they might give a negative answer although the given task set is in fact schedulable. If a *necessary* schedulability test gives a negative answer, then the task set is definitely not schedulable. For the class of exact schedulability tests, however, Garey and Johnson proved in their seminal work [79] that almost all classes of schedulability tests are **NP**-complete, i. e., computational intractable unless **P** = **NP**.

In contrast, a static scheduler makes its scheduling decisions at compile-time. Therefore, the scheduler generates a dispatching table offline, which is then used by the run-time dispatcher to initiate activities at predefined points in time. As the computation is done at compile-time, more sophisticated and complex algorithms can be applied and even different plans (tables) can be tried. In this scenario, everything is planned before the system is deployed. These algorithms need to have detailed prior knowledge about the deployed system. In particular the task

**static scheduling**

characteristics, e. g. worst-case execution times, precedence constraints, mutual exclusion constraints as well as deadlines, are of importance. The prior knowledge makes the systems predictable and deterministic, i. e., the developer knows exactly the system's behaviour over time. Since only dispatching tables have to be managed by the run-time dispatcher one observes only low run-time overhead. The computation of the dispatching table can be time-consuming, which, however, is affordable at compile-time. Unlike dynamic scheduling, which works for both periodic and aperiodic tasks, the strengths of static scheduling approaches are focused on periodic task invocations.

**hybrid scheduling**

Besides the two mentioned approaches, a combination of both—*hybrid* scheduling—is possible. In this setting, an offline schedule is computed for a subset of—mostly safety critical—tasks. The operating system's dispatcher executes those tasks accordingly and schedules additional tasks when the system is idle and not used for time-critical tasks.

Both approaches can either be *preemptive* or *non-preemptive*. Following the preemptive paradigm, a currently executing task can be interrupted as soon as another, more urgent task needs to be executed. On the other hand, non-preemptive scheduling prohibits task interruption. Table 7.2 summarises the most important properties.

| Static Scheduling | Dynamic Scheduling |
|---|---|
| + deterministic | − non-deterministic |
| + highly predictable | − limited predictable |
| + low run-time overhead | − high run-time overhead |
| − inflexible | + flexible |

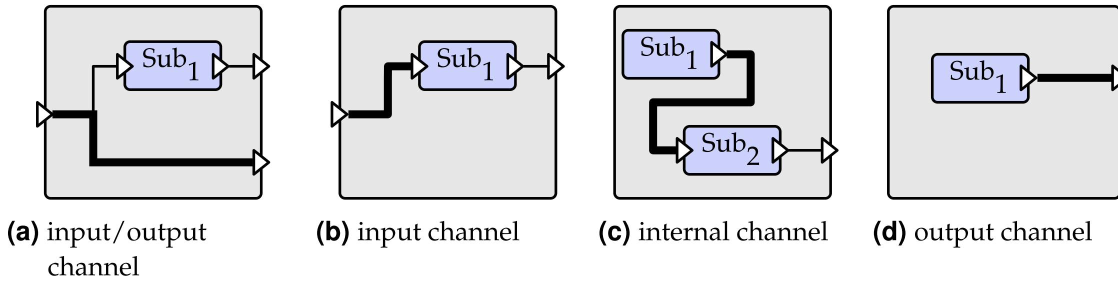
**Table 7.2.:** Comparison between static and dynamic hard real-time scheduling.

As mentioned before, one of the main advantages of the static (pre-run-time) approach is *predictability*. This thesis focuses on automotive systems, where predictability, reliability, and safety are desirable attributes. Therefore, we decided to sacrifice flexibility—in the case of dynamic scheduling—in favour of having full prior control and predictability, as far as the available pre-run-time information allows.

### 7.3.3. Dependency Analysis

Before a feasible schedule for the distributed COLA system can be performed, one has to determine cluster dependencies arising from data-flow dependencies and

operating modes. On level of the Cluster Architecture (cf. Section 4.2.3), the logical COLA model is cut into deployable atomic entities—the clusters. The underlying data-flow induced by channels between networks defines the dependencies. Data-flow is then given in the following cases depicted in Figures 7.8a to 7.8d with bold channels:




**(a)** input/output channel **(b)** input channel **(c)** internal channel **(d)** output channel

**Figure 7.8.:** Possibilities for data-flow in COLA networks: (a) input to output flow, (b) input to input flow, (c) output to input flow, and (d) output to output flow.

(i) Data-flow from an input port of a network to an output port of the same network,

(ii) between an input port of a network to an input port of a connected sub-unit,

(iii) between sub-units of a network, i. e., from a single output port to at least one input port of another sub-unit, and finally

(iv) between an output port of a sub-unit and an output port of the surrounding network.

Besides data-flow dependencies, the cluster dependency graph does also reflect control-flow dependencies induced by mode automata. Depending on the current mode of operation, only a subset of all possible clusters is executed, exactly those that are necessary for a correct functional service delivery.

Both data-flow and control-flow dependencies are important not only for scheduling, but also for code generation and system configuration. Hence, we introduced a data structure, namely the *Cluster Dependency Graph* [128].

**Cluster Dependency Graph (CDG)**

**Definition 9** (Cluster Dependency Graph)**.** *A Cluster Dependency Graph (CDG) is a directed, acyclic graph* $\mathcal{G} = \langle V_w, V_m, V_b, E_d, E_m \rangle$ *with three types of pairwise disjoint vertices:* working cluster vertices $V_w$*,* mode cluster vertices $V_m$*, and* buffer vertices $V_b$*. The set of clusters C of the Cluster Architecture is represented by* $V_w \cup V_m$*. The set*

*of edges is divided into* data-flow edges $E_d$ *and* mode edges $E_m$ *with* $E_d \cap E_m = \emptyset$. *It holds:* $E_d = \{(u,v) \mid u \in V_w, v \in V_b\} \cup \{(u,v) \mid u \in V_b, v \in (V_w \cup V_m)\}$ *and* $E_m = \{(u,v) \mid u \in V_m, v \in (V_w \cup V_m)\}$.

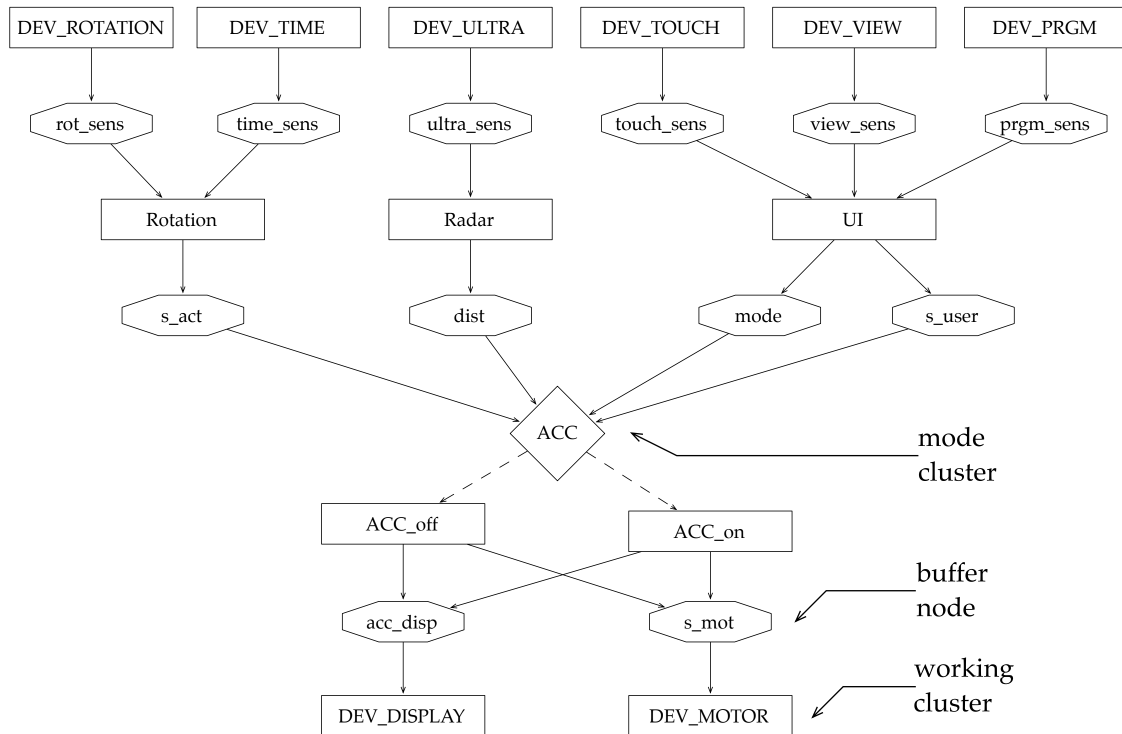



**Figure 7.9.:** Cluster dependency graph of the case study accomplished in Chapter 8.1.

The graphical representation of a cluster dependency graph is given in Figure 7.9. The depicted CDG is taken from the case study's model of Chapter 8.1. Working cluster nodes are depicted as rectangles, mode cluster vertices as diamonds, and buffer nodes as octagons. Solid edges symbolise that either values are written to or read from a buffer. Communication between clusters takes place whenever the model of the Logical Architecture is cut in a way that connected units are spread over different clusters. In this case, for each channel of the Logical Architecture traversing cluster boundaries on the Cluster Architecture, a buffer vertex is generated. Dashed arrows indicate exclusive mode alternatives, i. e., in the example either the mode 'ACC_on' or 'ACC_off' is active, but not both. 'ACC' decides based on its internal state and the incoming values which mode to chose. Note, 'ACC_on' and 'ACC_off' have no other ingoing edges, since it is implicitly clear from which buffers values are read, namely those pointing to 'ACC'.

The figure has a top-bottom layout, i. e., on top, there are sensors to be read indicated with the prefix 'DEV_' (device) followed by buffers storing the values which in turn are processed by the following working clusters. According to the current mode, alternative cluster sets are executed before writing the fresh data to buffers in order to be accessible for the actuators—again having the 'DEV_' prefix. As mentioned before, the employed middleware (cf. Figure 7.3) is used for namely two reasons:

(i) Transparent data communication and synchronisation.

(ii) Storage of the whole system state.

Each buffer node represents a concrete buffer in the middleware having a distinct logical address to be accessible. Moreover, each cluster has to store its internal state including delay values and automata states. All this is stored as a struct and accessible via a distinct logical address. Mode clusters in turn need two addresses for data storage: one for their internal state and one to store the current mode decision. The middleware API is given in Figure 7.10. These functions are used during C code generation to obtain the system time, read data from and write data to the middleware, respectively, and to save or restore a task's state. The cluster dependency graph is essential for the generation of schedule plans. This is done in two steps: first, for each possible mode, a set of clusters is generated. Second, for each of those, a linearisation is performed with respect to all dependencies. Together with determined worst-case execution times and deadlines, feasible schedules are computed.

#### Determine clusters to be scheduled for each mode

As can be seen in the CDG of Figure 7.9, not only the causal data-flow, but also the current operating mode determines the set of active clusters. This set has to be scheduled in a certain mode in order to fulfil the desired behaviour. This example has only one mode cluster that can be in two modes. Hence, one observes two operating modes with the sets:

$$
\begin{aligned}
ON = \{ \quad & DEV_ROTATION, DEV_TIME, DEV_ULTRA, DEV_TOUCH, \\
& DEV_VIEW, DEV_PRGM, Rotation, Radar, UI, ACC, \mathbf{ACC_on}, \\
& DEV_DISPLAY, DEV_MOTOR \quad \} \\
OFF = \{ \quad & DEV_ROTATION, DEV_TIME, DEV_ULTRA, DEV_TOUCH, \\
& DEV_VIEW, DEV_PRGM, Rotation, Radar, UI, ACC, \mathbf{ACC_off}, \\
& DEV_DISPLAY, DEV_MOTOR \quad \}
\end{aligned}
$$

```
/* send data to middleware, distribute to other nodes */
int mw_send(void *buf, ssize_t len, unsigned short int dataid);

/* get data by ID */
int mw_receive(void *buf, ssize_t len,
                                        unsigned short int dataid);

/* save data to middleware, no distribution */
int mw_save_task_state(void *buf, ssize_t len,
                                        unsigned short int taskid);

/* get saved data from middleware */
int mw_restore_task_state(void *buf, ssize_t len,
                                        unsigned short int taskid);

/* get global time for all nodes, time in nanoseconds will
   be stored in *time (uint64)                                  */
int mw_global_time(RTIME *time);
```

**Figure 7.10.:** API of the middleware developed by Haberl et al. [85].

This scenario, however, is the simplest. It becomes much more complicated when having hierarchies of modes, i. e., a mode cluster is decomposed again into an arbitrary number of sub-modes. Assume for example that a function decomposed into multiple modes has different behaviours depending on the active powertrain, namely, combustion engine, electric motor, or a hybrid setting.

A more complex setting with hierarchical modes is given in Figure 7.11a. We see the three mode cluster vertices C, D, I and the working clusters A, B, E, F, G, H, J, K, L, and M. For the sake of simplicity, the CDG is reduced by removing the buffer vertices, as they are not used during scheduling. Edges directing to buffer vertices are looped through it and now point to the original destination of the buffer's outgoing edges.

**Definition 10** (Reduced Cluster Dependency Graph)**.** *A* reduced *cluster dependency graph (RCDG)* $\mathcal{G}^r = \langle V_w, V_m, E \rangle$ *is obtained from a cluster dependency graph* $\mathcal{G} = \langle V_w, V_m, V_b, E_d, E_m \rangle$ *by doing the following: let* $v \in V_b$ *be a buffer node. Furthermore, let* $E_i$ *be the set of edges pointing to* $v$*, i. e.,* $E_i = \{(v', v) \mid v' \in V_w\}$ *and* $E_o$ *the set of*

*outgoing edges, i. e.,* $E_o = \{(v, v') \mid v' \in (V_w \cup V_m)\}$.

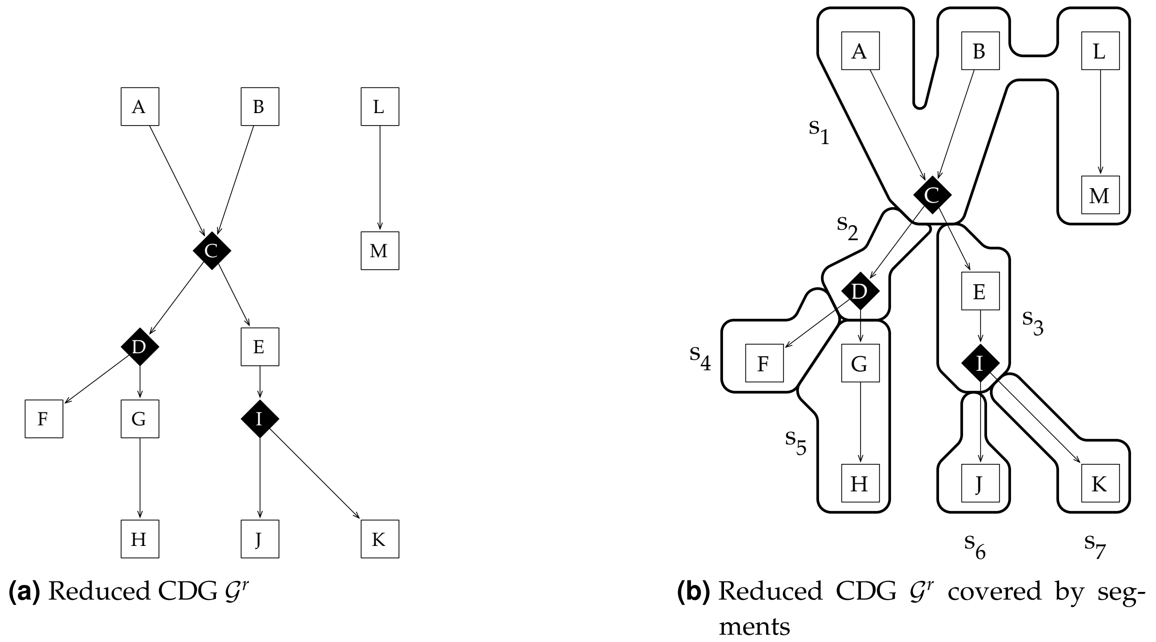
```
input  : CDG 𝒢 = ⟨V_w, V_m, V_b, E_d, E_m⟩
output : RCDG 𝒢^r = ⟨V'_w, V'_m, E⟩
1 V'_w ← V_w
2 V'_m ← V_m
3 E ← ∅
4 foreach v ∈ V_b do
5     foreach (v', v) ∈ E_i do
6         foreach (v, v'') ∈ E_o do
7             E ← E ∪ {(v', v'')}
8 E ← E ∪ E_m
9 return ⟨V'_w, V'_m, E⟩
```

*The outlined algorithm returns the reduced cluster dependency graph. It simply removes all buffer nodes* $v \in V_b$ *and creates for each pair of ingoing and outgoing edge* $(v', v)$ *and* $(v, v'')$*, respectively, a new edge* $(v', v'')$ *bypassing the removed buffer node. Note that the graphical representation of the RCDG does not distinguish between different edge types anymore.*

In Figure 7.11a a reduced CDG is depicted consisting only of working cluster and mode cluster vertices as well as one edge type.

**Definition 11** (Precedence relation $\prec$)**.** *The precedence relation is defined among clusters, i. e., the expression* $c_i \prec c_j$ *means that cluster* $c_i$ *precedes cluster* $c_j$*. In other words, the execution of cluster* $c_i$ *has to be finished before* $c_j$ *can be started. If two clusters have no relation, they are independent and thus can be executed in parallel if more than one execution node is available.*

The precedence relation provides information in which causal relation clusters can be executed. In the example of Figure 7.11a, A and B have to be finished before C can be started. L and M as a separate connected component are independent from all nodes in the connected component on the left, hence can in principle be executed in parallel (depicted with $\|$ in the table of Figure 7.11c). In Figure 7.11b the reduced CDG is covered by segments $s_1$ to $s_7$. A combination of these segments determines the set of clusters to be scheduled in a certain mode. The hierarchical combination of mode vertices induces four different node sets, each for one mode of execution. The computation of schedule sets involves a sophisticated treatment especially because of hierarchical modes.




**(a)** Reduced CDG $\mathcal{G}^r$

**(b)** Reduced CDG $\mathcal{G}^r$ covered by segments

| $\prec$ | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | $\square$ | $\parallel$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\parallel$ | $\parallel$ |
| B | | $\square$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\parallel$ | $\parallel$ |
| C | | | $\square$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\prec$ | $\parallel$ | $\parallel$ |
| D | | | | $\square$ | | $\prec$ | $\prec$ | $\prec$ | | | | $\parallel$ | $\parallel$ |
| E | | | | | $\square$ | | | | $\prec$ | $\prec$ | $\prec$ | $\parallel$ | $\parallel$ |
| F | | | | | | $\square$ | | | | | | $\parallel$ | $\parallel$ |
| G | | | | | | | $\square$ | $\prec$ | | | | $\parallel$ | $\parallel$ |
| H | | | | | | | | $\square$ | | | | $\parallel$ | $\parallel$ |
| I | | | | | | | | | $\square$ | $\prec$ | $\prec$ | $\parallel$ | $\parallel$ |
| J | | | | | | | | | | $\square$ | | $\parallel$ | $\parallel$ |
| K | | | | | | | | | | | $\square$ | $\parallel$ | $\parallel$ |
| L | | | | | | | | | | | | $\square$ | $\prec$ |
| M | | | | | | | | | | | | | $\square$ |

**(c)** Overview on the precedence relation and concurrent executions of the given example.

**Figure 7.11.:** Figure (a) shows a reduced cluster dependency graph, which can be covered by segments depicted in (b). The precedence relation and possible concurrent executions are illustrated in table (c).

**Definition 12** (Schedule Set)**.** *A schedule set $S \subseteq (V_w \cup V_m)$ is a finite set of clusters to be scheduled in order to adduce a mode's behaviour.*

The four modes have the schedule sets:

$$S_1 = s_1 \cup s_2 \cup s_4 = \{A, B, C, D, F, L, M\}$$
$$S_2 = s_1 \cup s_2 \cup s_5 = \{A, B, C, D, G, H, L, M\}$$
$$S_3 = s_1 \cup s_3 \cup s_6 = \{A, B, C, E, I, J, L, M\}$$
$$S_4 = s_1 \cup s_3 \cup s_7 = \{A, B, C, E, I, K, L, M\}$$

Let $\pi = c_1 \dots c_k$ be a path of length $k$ in the reduced cluster dependency graph. Furthermore let $\Pi = \{\pi_1, \dots, \pi_m\}$ be the set of all paths from root vertices, i. e., vertices without ingoing edges, to leaf vertices, i. e., nodes without outgoing edges. Then, Algorithm 7 will output a set of schedule sets $\mathbf{S}$.

The algorithm works as follows: first of all, in line 2 a depth-first search in the reduced CDG $\mathcal{G}^r = \langle V_w, V_m, E \rangle$ is performed to obtain all paths from root to leaf nodes, yielding $\Pi$. In a second step (cf. lines 3 to 8), all path prefixes consisting only of working cluster vertices are removed and their elements are added to an initial set $S_{init}$. Moreover, this set also includes the head elements of the shortened paths, i. e., mode cluster nodes. The initial schedule set $S_{init}$ contains all clusters that are executed in any case, i. e., no matter which operating mode is active. For the running example this is $S_{init} = \{A, B, C, L, M\}$. After generating the initial schedule set, the algorithm continues to iterate over the path set $\Pi$ until all paths have been processed, i. e., have length zero (cf. line 9). Depending on the type of the head element at hand (working or mode cluster node, cf. lines 11 and 18) the procedure differs:

(i) **Working cluster node** $v_w = hd(\pi) \in V_w$: If the length of the current path $|\pi|$ is grater than one, we add the second element of $\pi$, i. e., $hd(tail(\pi))$ to that schedule set $S \in \mathbf{S}$ already containing the node $v_w$. Otherwise, no more elements have to be added. In both cases the procedure `removePrefix(`$v_w$`)` is called. This procedure iterates over all paths $\pi \in \Pi$ and removes the prefix $v_w$ if possible.

(ii) **Mode cluster node** $v_m = hd(\pi) \in V_m$: Mode cluster nodes introduce a branching of schedule sets. Each already computed schedule set $S \in \mathbf{S}$ containing the mode cluster node $v_m$ is replaced by $c$ schedule sets derived from $S$ but each being extended by the respective nodes for each mode, i. e., $c = |\{v \in (V_m \cup V_w) \mid (v_m, v) \in E\}|$. Again, `removePrefix(`$v_m$`)` is called.

**input** : Reduced CDG $\mathcal{G}^r = \langle V_w, V_m, E \rangle$
**output** : Set of schedule sets $\mathbf{S}$

1 $\mathbf{S} \leftarrow \varnothing$
2 Traverse $\mathcal{G}^r$ to obtain the set of paths $\Pi$
/* build initial schedule set $S_{init}$ */
3 $S_{init} \leftarrow \varnothing$
4 **foreach** $\pi \in \Pi$ **do**
5     **while** $hd(\pi) \neq nil$ **do**
6         $S_{init} \leftarrow S_{init} \cup \{hd(\pi)\}$
7         **if** $hd(\pi) \in V_m$ **then break else** $\pi \leftarrow tail(\pi)$
8 $\mathbf{S} \leftarrow \mathbf{S} \cup \{S_{init}\}$
/* continue constructing the set of schedule sets $\mathbf{S}$ */
9 **while** $\exists\, \pi \in \Pi \;:\; |\pi| > 0$ **do**
10     choose such a $\pi$
    /* processing working node */
11     **if** $hd(\pi) \in V_w$ **then**
12         **if** $|\pi| > 1$ **then**
13             **let** $S \in \mathbf{S} : hd(\pi) \in S$
14             $S \leftarrow S \cup \{hd(tail(\pi))\}$
15             `removePrefix(`$hd(\pi)$`)`
16         **else** `removePrefix(`$hd(\pi)$`)`
17     /* processing mode node with branching */
18     **else if** $hd(\pi) \in V_m$ **then**
19         **foreach** $S \in \{S' \in \mathbf{S} \mid hd(\pi) \in S'\}$ **do**
20             **foreach** $v \in \{v' \in (V_m \cup V_w) \mid (hd(\pi), v') \in E\}$ **do**
21                 $S' \leftarrow S \cup \{v\}$
22                 $\mathbf{S} \leftarrow \mathbf{S} \cup \{S'\}$
23             $\mathbf{S} \leftarrow \mathbf{S} \setminus \{S\}$
24             `removePrefix(`$hd(\pi)$`)`
25 **return** $\mathbf{S}$
26 **procedure** `removePrefix(`$v \in (V_w \cup V_m)$`)`
27     **foreach** $\pi \in \Pi$ **do**
28         **if** $|\pi| > 0$ & $hd(\pi) == v$ **then** $\pi \leftarrow tail(\pi)$

**Algorithm 7:** Returns a set of schedule sets $\mathbf{S}$, i. e., each of the sets contains those clusters to be scheduled to deliver an operating mode's service.

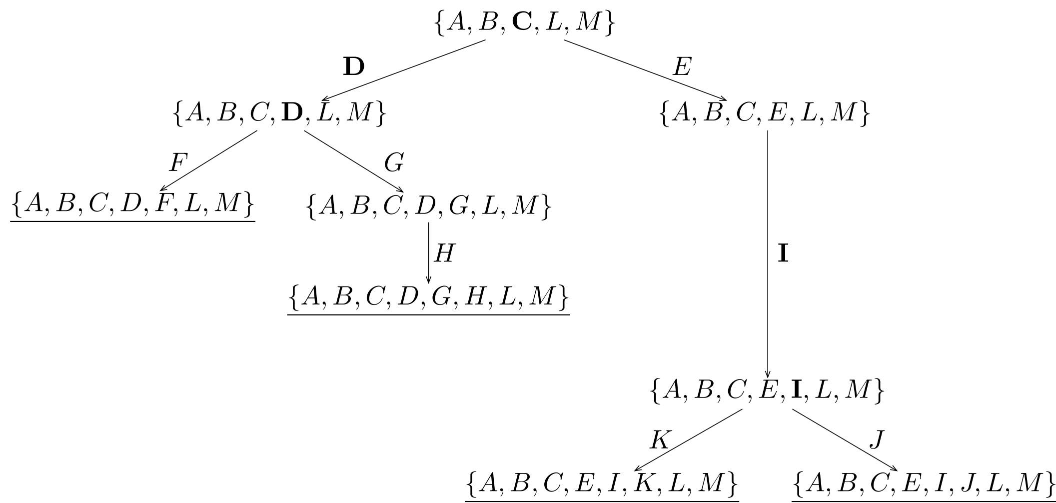



**Figure 7.12.:** Exemplary run of Algorithm 7 using the reduced CDG depicted in Figure 7.11a. Π as listed below was processed from top to bottom:

```
(1) ACDF   (6) BCDGH
(2) ACDGH  (7) BCEIJ
(3) ACEIJ  (8) BCEIK
(4) ACEIK  (9) LM
(5) BCDF
```

Termination of the algorithm is guaranteed because there is only a finite number of paths within the reduced cluster dependency graph. At each iteration of the **while**-loop the paths are reduced in their length, finally satisfying the termination condition of the loop (cf. line 9).

Figure 7.12 visualises an exemplarily run of the algorithm using the running example. Arrows are labeled with the element added to the next lower level. Bold letters indicate mode cluster vertices introducing a branching of schedule sets.

Together with the precedence relation $\prec$, which can directly be derived from the reduced cluster dependency graph, the correct cluster order can be determined.

**Definition 13** (Schedule List)**.** *A schedule list $\mathcal{L} = \langle S, \prec \rangle$ is a finite sequence $l$ with $|l| = |S|$ respecting the precedence relation $\prec$. $l$ is a permutation of $S$.*

**Definition 14** (Timed Schedule List)**.** *A* timed *schedule list $\mathcal{L}^\tau = \langle S, \prec, \tau \rangle$ is a schedule list $\mathcal{L}$ where each element $c \in S$ is assigned a starting time using $\tau : S \to \mathbb{N}_0^+$.*

A possible schedule list for $S_3$, i. e., $\langle S_3, \prec \rangle$ would be for example: *ALBCEIMJ*. Now, we know which clusters to schedule in which mode and also their causal order. In real-time systems, however, the correct point in time *when* to schedule a task or cluster is of importance. The correct order is thus insufficient.

To meet the requirements of real-time systems, deployment of the COLA automotive approach includes the determination of the worst-case execution time of clusters with respect to the target hardware they are executed on (cf. also step **S3** of Section 4.4).

**Definition 15** (Schedule)**.** *A* schedule *for the complete* COLA *system is determined by the triple $\mathscr{S} = \langle \mathcal{G}^r, \alpha, T \rangle$. $\mathcal{G}^r$ is the reduced cluster dependency graph, $\alpha$ the allocation decision, and $T$ a finite set of temporal constraints like e. g. WCET and deadlines. If there is a schedule, then it satisfies the precedence relation $\prec$ induced by $\mathcal{G}^r$ and the temporal constraints $T$.*

In the following, constraints are listed that encode the circumstance of temporal ($T$) and precedence properties ($\prec$) of distributed COLA systems.

### 7.3.4. Constraint System

For hard real-time systems, a timely execution is fundamental for a correct and safe operation. Therefore, several general constraints have to be satisfied by a static scheduler in order to guarantee these strict requirements. Only when the points in time, when clusters (tasks) are activated by a dispatcher are correct, the system operates as desired. In the following, some general constraints that have to be fulfilled are given. A first distinction is made between single-ECU and multi-ECU systems, as independent clusters can be executed in parallel on multi-ECU systems, which—of course—is prohibited in single-ECU systems. Therefore, the constrains encode amongst others the precedence relation and temporal attributes.

#### Generic constraints

Below, generic constraints are listed that have to be satisfied in both the single and the multi-ECU case.

**(G1) Wait for sensor values.** As the COLA evaluation follows the IPO-model, the processing phase follows the input phase, thus it holds $\forall\, 1 \leq i \leq n$

$$s_{i,j} > \iota \qquad j \in \mathbb{N}^+ \tag{7.16}$$

where $\iota$ is the time needed to read the inputs.

**(G2) Consider time to write actuators.** Again, the IPO-model states that the output phase follows the processing phase, thus it holds $\forall\, 1 \leq i \leq n$

$$f_{i,j} < \gamma - o \qquad j \in \mathbb{N}^+ \tag{7.17}$$

where $o$ is the time needed to write the outputs and $\gamma$ is the cycle time. That means, we implicitly assume for all jobs the deadline $D_i$ to be $\gamma - o$.

**(G3) Worst-case execution time.** We assume that all jobs $c_{i,j}$ have the same worst-case execution time $C_i$. As the worst-case execution time $C_i$ may be different depending on the executing processor, the allocation $\alpha$ has to be considered, too. For the finishing times it holds $\forall\, 1 \le i \le n$

$$f_{i,j} = s_{i,j} + C_i^p \qquad j \in \mathbb{N}^+ \tag{7.18}$$

where $C_i^p$ denotes the worst-case execution time of cluster $c_i$ on processor $p$ assuming $\alpha(c_i) = p$. $C_i^p$ is derived from $\mathscr{R}$, cf. also Section 7.2.1.

### Single-ECU system

As in *single-ECU* systems, the execution of tasks has to be linearised according to the topological order, which is induced by the data-flow dependencies, thus no concurrent execution is allowed.

**(S1) Non-overlapping condition.** No two jobs $c_{s,j}$ and $c_{t,j}$ are allowed to overlap, i. e., it holds $\forall\, 1 \le s, t, \le n,\ s \ne t$

$$\big((s_{s,j} > f_{t,j}) \wedge (\neg(s_{t,j} > f_{s,j}))\big) \vee \big((s_{t,j} > f_{s,j}) \wedge (\neg(s_{s,j} > f_{t,j}))\big) \qquad j \in \mathbb{N}^+ \tag{7.19}$$

**(S2) Precedence relation.** If two jobs $c_{s,j}$ and $c_{t,j}$ are interdependent, i. e., $c_{s,j} \prec c_{t,j}$ then it holds $\forall\, 1 \le s, t, \le n,\ s \ne t$

$$f_{s,j} < s_{t,j} \qquad j \in \mathbb{N}^+ \tag{7.20}$$

### Multi-ECU system

The usual case, and also the case primarily considered by the COLA automotive approach is the multi-ECU setting. These systems have multiple ECUs realising a single function, or a set of functions. These can be completely independent of each other, or—which is the normal case for the automotive domain—are to a high degree coupled and therefore interdependent. An appropriate consideration of multi-ECU systems thus needs to regard the placement of clusters—the allocation.

**(M1) Non-overlapping condition.** No pair of *independent* jobs allocated onto the same ECU is allowed to overlap, i. e., it holds $\forall\, 1 \le s, t, \le n,\ s \ne t$ with $\alpha(c_s) = \alpha(c_t)$

$$\big((s_{s,j} > f_{t,j}) \wedge (\neg(s_{t,j} > f_{s,j}))\big) \vee \big((s_{t,j} > f_{s,j}) \wedge (\neg(s_{s,j} > f_{t,j}))\big) \qquad j \in \mathbb{N}^+ \tag{7.21}$$

**(M2) Precedence relation.** If two jobs $c_{s,j}$ and $c_{t,j}$ are *interdependent*, i. e., $c_{s,j} \prec c_{t,j}$ then it holds $\forall\, 1 \leq s, t, \leq n,\ s \neq t$

$$f_{s,j} < s_{t,j} \qquad j \in \mathbb{N}^{+} \tag{7.22}$$

### Extension to Multi-Rate Scheduling

**multi-rate scheduling**

A possible extension, which is currently not implemented in the COLA engineering environment, is *multi-rate scheduling*. It is defined as a set of periodic tasks (clusters) where the number of jobs per cycle may vary from cluster to cluster, i. e., some clusters are executed more often than others within the same cycle. In this case, one does not longer speak about a cycle but about a *hyper-period* $H$. The hyper-period is defined to be the least common multiplier of the task periods

$$H = \operatorname*{lcm}_{1 \leq i \leq |C|} \{T_i\}\,. \tag{7.23}$$

For the EDF algorithm, Leung and Merrill [138] proved that it is sufficient to consider the hyper-period, as it defines the shape of all future executions. This assumes that the release times are equal for all clusters, i. e., $R_1 = R_2 = \cdots = R_n$ where $n$ is the number of clusters. If, however, it holds $R_i \neq R_j$ for some $1 \leq i, j \leq n$, then they showed that the time horizon to consider is no more the hyper-period, but a new bound $H'$ defined as

$$H' = \underset{1 \leq i \leq |C|}{max} \{R_i\} + 2H. \tag{7.24}$$

Hence, if all necessary and sufficient conditions for a dynamic scheduling algorithm, such as EDF, are fulfilled, the outlined approach will find a valid scheduling plan, as well. The presented approach can be considered as a generic approach, i. e., if there is any scheduling algorithm finding a feasible schedule, then we will find it, too.

Assume for example the four clusters with arbitrarily chosen values for the attributes period $T_i$, deadline $D_i$, and WCET $C_i$ listed in Table 7.3. Figure 7.13 shows the job invocations of the four clusters. Arrowheads indicate deadlines of the respective jobs. In this example no precedence relation is considered. However, in combination with the already stated constraints for multi-ECU systems, this is accomplished. The hyper-period defines the amount of time that has to be considered during analysis. This is sufficient, since job invocations continuously repeat after the first hyper-period. The number of job invocations $\#_i$ of cluster $c_i$ per hyper-period $H$ is determined by the period $T_i$ of a cluster.

$$\#_i = \frac{H}{T_i} \quad 1 \leq i \leq |C| \tag{7.25}$$

| Cluster | Period [ms] | Deadline [ms] | WCET [ms] |
|---|---|---|---|
| Speed measurement | 60 | 60 | 40 |
| ESP | 20 | 20 | 7 |
| Fuel injection | 10 | 10 | 3 |
| ABS | 40 | 40 | 10 |

**Table 7.3.:** Multi-rate periodic cluster scheduling. Clusters have different periods and deadlines.

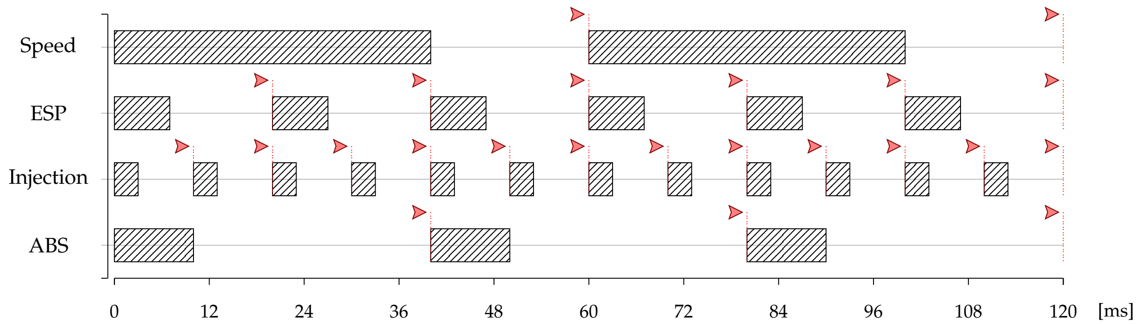



**Figure 7.13.:** Multi-rate scheduling example. $H = 120$ ms.

**(R1) Deadlines.** The deadlines for each job $d_{i,j}$ within the hyper-period $H$ are described by

$$d_{i,j} = ((j-1) \bmod \#_i + 1)\, D_i \qquad j \in \mathbb{N}^+ \tag{7.26}$$

**(R2) Start times.** The constraint **G1** has to be changed in a way that it considers multiple jobs within a hyper-period.

$$s_{i,j} \geq ((j-1) \bmod \#_i)\, T_i \tag{7.27}$$

$$s_{i,j} \leq ((j-1) \bmod \#_i + 1)\, T_i) - C_i \tag{7.28}$$

For $j \in \mathbb{N}^+$.

## 7.3.5. Realisation

In the previous section, numerous constraints were given, which a feasible schedule has to satisfy, in order to guarantee preservation of the COLA semantics down to the execution platform. Depending on the structure of operating modes, a quite large number of hierarchically nested modes is possible. As for each nested

mode, the schedule set is forked, it is essential that a schedule for each of them is computed efficiently. For the example given in Figure 7.11a one can assume for example that node C distinguishes between combustion engine and electric motor (E) and then whether the seat heater works in 'ECO' mode, i. e., with reduced voltage, or is disabled at all if the power supply is not sufficient. Electric mobility and its consequences concerning functional realisation is a very good example where the concept of operation modes takes full effect.

**SMT solver**

For fast and effective schedule enumeration, the power of available SMT solvers is used, in particular the public available YICES [63] solver. Given the COLA system scheduling problem $\mathscr{S}$, the solver results *a single* feasible schedule, if there is one. The COLA automotive approach pursues the goal of minimising the makespan, i. e., the minimisation of the finishing time of the last cluster execution of each cycle (cf. also minimum makespan scheduling). The obtained result, however, is not guaranteed to be optimal in the mentioned sense. As most SMT solvers do not optimise as for instance solvers for linear programming where an optimisation goal can be given, such a result has to be searched for. For this purpose, we gradually prune the search space until no satisfying result can be found anymore. Hence, the last found is also the optimal one. Therefore, a binary search is performed within bounds that encode both the best- and the worst-case (lower bound *lb*, upper bound *ub*), i. e., all clusters are aligned left-most and right-most, respectively, with respect to the schedule cycle. Figure 7.14 gives an example with four clusters $c_1$ to $c_4$ with the worst-case execution times $C_1 = 10$, $C_2 = 20$, $C_3 = 30$, and $C_4 = 40$. The cycle time $\gamma$ is assumed to be 150. Fifty time units, say milliseconds, are unused. In the upper part of the figure, the best-case is

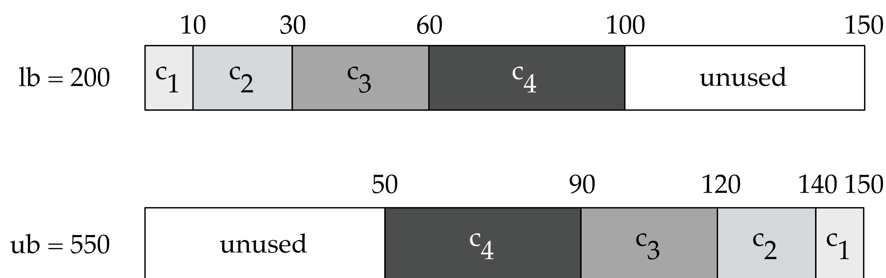



**Figure 7.14.:** Determination of initial bounds

depicted, i. e., all clusters are aligned left-most. Contrariwise, in the lower part, all clusters are aligned right-most. Between these extrema, a feasible schedule for the complete system can be found. Note, in the following formulae, the precedence constraint $\prec$ and temporal constrains $T$ are not considered, as they are part of the

remaining constraint system.

To render the search operational, the bounds are computed as follows:

$$lb = \sum_{p \in P} \sum_{i=1}^{|\alpha^{-1}(p)|} \sum_{j=1}^{i} C_j^p \tag{7.29}$$

$$\begin{aligned} ub &= \sum_{p \in P} \sum_{i=1}^{|\alpha^{-1}(p)|} \left( \gamma - \sum_{j=1}^{|\alpha^{-1}(p)|} C_j^p + \sum_{k=0}^{i-1} C_{|\alpha^{-1}(p)|-k}^p \right) \\ &= \sum_{p \in P} \left( \gamma + \sum_{i=1}^{|\alpha^{-1}(p)|-1} \left( \gamma - \sum_{j=1}^{i} C_j^p \right) \right) \end{aligned} \tag{7.30}$$

where $C_i^p$ denotes the worst-case execution time of cluster $c_i$ executed on ECU $p$. The bounds accrue from the sums of finishing times of all clusters allocated on the different ECUs.

**Definition 16** (Optimal Schedule)**.** *Let $\mathscr{S} = \langle \mathcal{G}^r, \alpha, T \rangle$ be a given scheduling problem for the complete* COLA *system. If the makespan for all possible execution modes is desired to be minimal, $\mathscr{S}$ then describes an* optimal *schedule problem, denoted by $\mathscr{S}^*$.*

In the following, only feasible solutions for the optimal scheduling problem $\mathscr{S}^*$ are of interest. On the one hand, when sorting the clusters in ascending order with respect to their worst-case execution times, and aligning them left-most, the contribution to $lb$ is minimised. On the other hand, when aligning the descending ordered clusters right-most, their contribution to $ub$ is maximised. In this vein, $lb$ and $ub$ are safe search bounds.

For the sake of simplicity the set of clusters allocated onto an ECU $p$, i. e., $\alpha^{-1}(p)$, is assumed to be indexed in a way that it consists of elements $c_1, \ldots, c_{|\alpha^{-1}(p)|}$. Actual clusters are mapped to the new indices, which is omitted here. For example if clusters $c_4$, $c_7$, and $c_2$ are mapped onto ECU $p$, $\alpha^{-1}(p) = \{c_1, c_2, c_3\}$ with an internal mapping $c_4 \mapsto c_1$, $c_7 \mapsto c_2$, $c_2 \mapsto c_3$, respectively. This is done ascending with respect to the WCET, i. e., $C_1^p \leq C_2^p \leq \ldots \leq C_{|\alpha^{-1}(p)|}^p$. Coming back to the example of Figure 7.14 the following bounds are obtained:

$$\begin{aligned} lb &= 10 + 30 + 60 + 100 && = 200 \\ ub &= 90 + 120 + 140 + 150 && = 500 \end{aligned}$$

Assuming there is a feasible solution for $\mathscr{S}$, i. e., satisfying for each mode the constraint system given in Section 7.3.4. Then there is also a feasible schedule where the sum of all cluster finishing times is within the interval $[lb, ub]$, i. e., adding the constraint

**(O) Optimisation condition.**

$$\sum_{i=i}^{|C|} f_{i,j} \leq b \qquad j \in \mathbb{N}^{+} \tag{7.31}$$

where $b \in [lb, ub]$ holds, does not change the satisfiability of the constraint system for at least one such $b$. If there is a minimal $b$ for every operating mode, a solution for the optimal schedule $\mathscr{S}^{*}$ has been found. If the system of constraints is unsatisfiable, then either the value of $b$ is too small, or the system would have been unsatisfiable even without the optimisation condition. In other words, a solution for $\mathscr{S}$ implies a solution for $\mathscr{S}^{*}$.

The aim of the scheduling algorithm is to find the minimal starting times (and therefore minimal finishing times) for all clusters of all operating modes, i. e., for each of them a minimal $b$ satisfying the constraint system. Algorithm 8—based on binary search—is used to find such a minimal $b$.

For each schedule set $S \in \mathbf{S}$, the algorithm tries to find a minimal $b$ such that all constraints are satisfied. Therefore, a binary search between the bounds $lb$ (lower bound) and $ub$ (upper bound) is performed.

The function `check` $(S, b)$ (cf. line 9) checks whether for the schedule set $S$ and the bound $b$ all constraints are satisfied. In the positive case, a timed schedule list $\mathcal{L}^{\tau} = \langle S, \prec, \tau \rangle$ for that particular mode determined by $S$ is retuned, otherwise *nil*. In particular, $\tau$ is calculated, i. e., the points in time when clusters have to be executed. If for all operating modes $\mathscr{S}^{*}$ has a non-*nil* result (cf. lines 19 and 20), the result for the COLA system scheduling problem is a set of timed scheduling lists, i. e., $\boldsymbol{\mathcal{L}}^{\tau}$.

In this vein, the earliest possible placement taking allocation $\alpha$ and data-flow dependencies $\prec$ into account is accomplished. The function `check` generates an input file for the YICES SMT-solver, which in turn checks whether there is a feasible schedule plan (timed schedule list) for the value $b$ and the given clusters $S$. Figure 7.15 visualises the convergence steps of the algorithm. The example shows the three modes 'normal', 'parking', and 'sdc_active' of the case study explained in Chapter 8.2. For the sake of clarity, the upper (1392) and lower (888) bounds are after the first and the second iteration of the described algorithm.

### 7.3.6. Complexity Analysis

The search for the minimal $b$s of each operating mode leads to a left-most alignment of all clusters in order to minimise the makespan. Therefore, the search is

**input** : Optimal Scheduling problem $\mathscr{S}^* = \langle \mathcal{G}^r, \alpha, T \rangle$
**output**: Solution for $\mathscr{S}^*$ if there is one, i. e., $\boldsymbol{\mathcal{L}}^\tau$, otherwise $nil$

1. $lb = \sum_{p \in P} \sum_{i=1}^{|\alpha^{-1}(p)|} \sum_{j=1}^{i} C_j^p$
2. $ub = \sum_{p \in P} \left(\gamma + \sum_{i=1}^{|\alpha^{-1}(p)|-1} \left(\gamma - \sum_{j=1}^{i} C_j^p\right)\right)$
3. $counter = 0$
4. **foreach** $S \in \mathbf{S}$ // $\mathbf{S}$ is obtained using Algorithm 7
5. **do**
6.     $\mathcal{L}^\tau_{last} \leftarrow nil$
7.     **while** $lb <= ub$ **do**
8.         $b = lb + \left(\frac{ub-lb}{2}\right)$
9.         $\mathcal{L}^\tau_{check} = \texttt{check}\,(S, b)$
10.         **if** $\mathcal{L}^\tau_{check} = nil$ **then**
11.             $lb \leftarrow b + 1$
12.         **else**
13.             $ub \leftarrow b - 1$
14.             $\mathcal{L}^\tau_{last} \leftarrow \mathcal{L}^\tau_{check}$
15.     **if** $\mathcal{L}^\tau_{last} \neq nil$ **then**
16.         $counter \leftarrow counter + 1$
17.         $\boldsymbol{\mathcal{L}}^\tau \leftarrow \boldsymbol{\mathcal{L}}^\tau \cup \{\mathcal{L}^\tau_{last}\}$
18.     **else**
19.         **return** $nil$
20. **return** $counter \neq |\mathbf{S}|\ ?\ nil\ :\ \boldsymbol{\mathcal{L}}^\tau$

**Algorithm 8:** Calculates a solution for the optimal scheduling problem $\mathscr{S}^*$ if there is one.

performed within the interval $[lb;\ ub]$. For each operating mode, searching needs $\log_2(ub - lb)$ steps in the worst-case. Let $m$ be the number of ECUs and $n$ the number of clusters to be scheduled, respectively. Without loss of generality, assume the WCET $C_i$ of all clusters is set to 1, i. e., $C_1 = C_2 = \ldots = C_n = 1$ no matter on which ECU a cluster is executed, and all clusters are uniformly distributed over the ECUs. Moreover we assume that $n$ is a multiple of $m$. The following analysis distinguishes between the lower and the upper bound for the search—beginning with the first one.

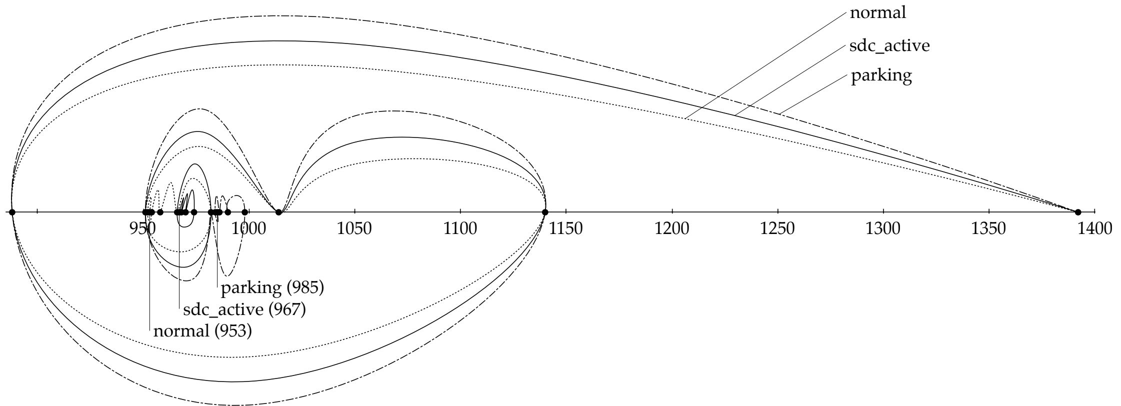



**Figure 7.15.:** The search converges for every mode to a minimal $b$ satisfying the constraint system.

### Lower bound

The earliest finishing times of all clusters is reached, when they start as early as possible within the cycle, i. e., aligned left-most. As we assumed the WCET to be one for all clusters, we do not need to order them ascending according to their WCETs. Consequently, the first cluster will finish at time point 1, the second at 2, and the last at $\frac{n}{m}$. Thus, the sum of all finishing times is as follows (summed over all ECUs):

$$lb = m \sum_{i=1}^{\frac{n}{m}} i = \frac{1}{2} \frac{n(n+m)}{m} \tag{7.32}$$

### Upper bound

The latest finishing times arise when the clusters are aligned right-most. Then, their finishing times are $\gamma, \gamma - 1, \ldots, \gamma - (\frac{n}{m} - 1)$, and in sum over all ECUs

$$ub = m \sum_{i=0}^{\frac{n}{m}-1} \gamma - i = \frac{1}{2} \frac{n(2\gamma m - n + m)}{m} \tag{7.33}$$

Both together yield the search space and thus the possible values for $b$, namely

$$ub - lb = m \left( \left( \sum_{i=0}^{\frac{n}{m}-1} \gamma - i \right) - \sum_{i=1}^{\frac{n}{m}} i \right) = n \frac{\gamma m - n}{m} \tag{7.34}$$

Hence, in the worst-case we expect

$$\log_2\left(n\frac{\gamma m - n}{m}\right) \tag{7.35}$$

calls of `check` (cf. line 9) for each operating mode. Of course, we are assuming $m, n$ to be positive and that the number of clusters per ECU is at most the cycle time, i. e., $\frac{n}{m} \leq \gamma$.

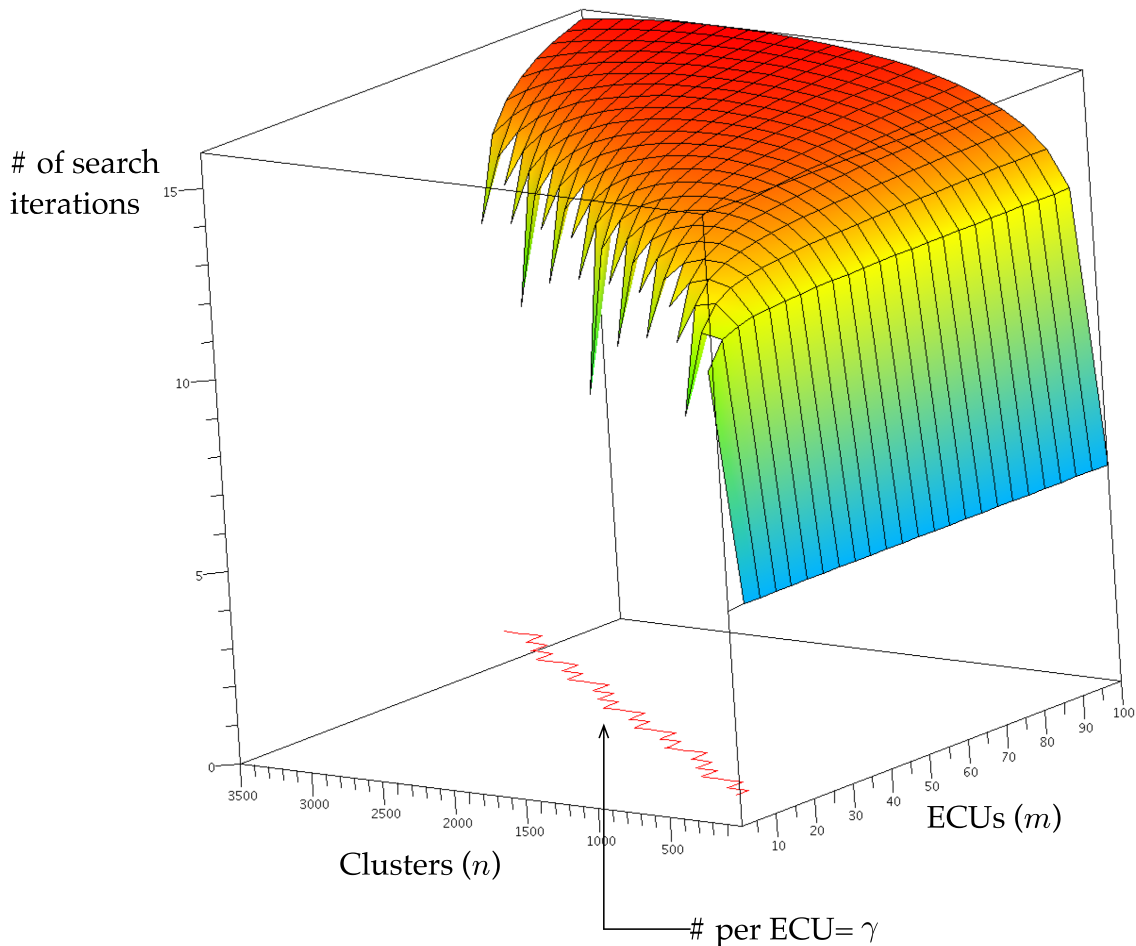



**Figure 7.16.:** Visualisation of the number of search iterations depending on the number of clusters $n$ and the number of ECUs $m$, with $\gamma = 50$.

## 7.4. Fault Tolerance

The COLA automotive deployment methodology guarantees—by construction—that the semantics of models on level of the Logical Architecture is preserved

down to the Technical Architecture and finally to the running target system.

> "The physical world is neither precise nor reliable, so why should we demand this of computing systems? Instead, we must make the systems robust and adaptive, building reliable systems out of unreliable components." [135, 136]

Following Lee, the subsequent explanations describe changes to the COLA automotive approach in order to make the therewith modelled systems even more robust and adaptive. Special *fault tolerant* modes are automatically generated, capable to cope with some system failures described below.

Therefore, first the fault model is described in Section 7.4.1 and then necessary adoptions in Section 7.4.2

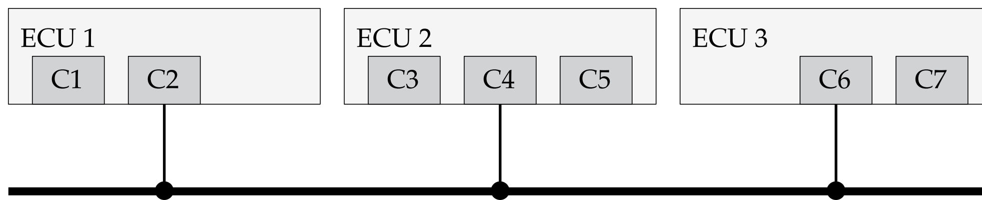
### 7.4.1. Fault Hypothesis

The intention of fault tolerant modes guarantees continuous operation of highly safety-critical clusters in case of hardware failures. To do so, the presented approach uses the already introduced concept of operating modes (cf. Section 4.3.2 and [88]). The fact that operating modes are computed offline, i. e., at design time, allows to switch them and hence their corresponding schedule plans easily at run-time. These differ in terms of contained clusters and their respective starting times. By relying on the time-triggered paradigm, determinism at run-time is guaranteed.

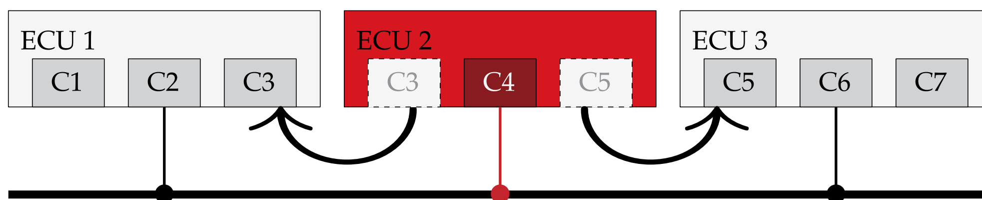
As spare capacity is rare in automotive E/E architectures, it is impossible to compensate an arbitrary number of ECU failures or bus breakdowns. Rather it is intended to guarantee the substituted execution of a few but most important clusters from a single failing ECU on another ECU containing spare resources. The decision about which clusters to select for redundancy is up to the developer and shall be specified within the model. This information in combination with the hardware platform model leads to a suitable failover ECU selection.

The concept outlined here is able to deal with the complete loss of a single ECU. While this case seems to be very special, it is quite common in practice. In an automotive system the ECUs as well as their connecting cables are exposed to permanent vibrations and changing environmental conditions like temperature, humidity, etc. These conditions can easily lead to broken wires, either in form of copper wires or the even more sensitive fibre optic cables, which may be used for example in MOST bus topologies. Other defects are loose contacts on or cracks in the employed circuit boards. While the described approach does not

address problems like a babbling idiot at the moment, a safe operating state can be guaranteed for the typical case of a not responding ECU, may it be because of communication loss, power loss, or loose electronic components. Figure 7.17




**(a)** Three ECUs are connected via a common bus. Clusters C1 to C7 are executed in a distributed setting.




**(b)** Either ECU 2 fails of the connection to the bus breaks down. Re-allocation of important clusters.

**Figure 7.17.:** Fault tolerance through re-allocation of clusters: (a) depicts the initial setting with three ECUs, 7 distributed clusters, and a shared bus. (b) Either ECU 2 fails or the link to the bus breaks down. Clusters C3 and C5 are re-allocated to ECUs 1 and 3, respectively. C4 is assumed to be non-critical and thus stays on ECU 2.

exemplifies the transition to a fault tolerant mode. In the upper Figure 7.17a, three ECUs and their contained clusters C1 through C7 are illustrated. In the lower part 7.17b a possible fault tolerant mode is shown, which is activated by the failure of ECU 2. In the exemplary fault tolerant mode clusters C3 and C5 are necessary for a safe operation of the system. Thus they are executed on ECU 1 and ECU 3 respectively, if ECU 2 fails. Cluster C4 is not safety-critical and thus is ignored in the fault tolerance mode.

One important constraint is that sensors and actuators that are needed by safety-critical clusters are either directly connected to the bus, or have a redundant connection to the failover ECUs. Otherwise they would not be accessible in case the ECU they are connected to fails, and thus the according cluster could not continue its operation successfully.

### 7.4.2. Adaptions

When considering the hardware deficiencies ECU failure, cable break, or signal loss one can in principle distinguish between

(i) *complete functioning* and

(ii) *partial functioning guarantee.*

The former is achieved by marking all clusters on an ECU as redundant if at least a single one is. Whereas the latter principle is also provided by ECUs with a mixture of redundant and non-redundant clusters. Regarding the cost pressure of the automotive domain in general and E/E systems in particular, it is appropriate to only concentrate on what is absolutely necessary. Thus in the following, the partial functioning guaranteed is favoured. If there were lots of spare hardware resources available, the first guarantee could also have been realised.

#### Allocation

From a set-theoretic point of view, the inverse mapping function $\alpha^{-1}: P \rightarrow 2^C$ can be interpreted as a partitioning of the set of clusters $C$ in $r \leq m = |P|$ disjoint non-empty parts $D_1, \ldots, D_r$—one for each used ECU defined in the Hardware Topology of the Technical Architecture. Each part encapsulates those clusters executed on the same ECU. Hence,

$$C = \biguplus_{1 \leq i \leq r} D_i = D_1 \uplus \cdots \uplus D_r \tag{7.36}$$

holds. When taking fault tolerant redundancy into account, a special marked cluster—be it because of realising a particularly important function—has to be deployed onto two different ECUs. In the standard mode (faultless execution) clusters are executed on the default ECU and their replica are unused, whereas in the fault-tolerant mode (in case of an error) the replicas are activated on the failover ECUs. Let

$$\mathcal{A} = \left\{ \overbrace{\{c_1, \ldots, c_e\}}^{\alpha^{-1}(p_1)=D_1}, \overbrace{\{c_{e+1}, \ldots, c_g\}}^{\alpha^{-1}(p_2)=D_2}, \ldots, \overbrace{\{c_l, \ldots, c_n\}}^{\alpha^{-1}(p_r)=D_r} \right\} \tag{7.37}$$

be a partition of the set of clusters C where $c_i \in C$ with $1 \leq i \leq n = |C|$ holds. We write $\widehat{c}$ to indicate that cluster $c$ is to be allocated redundantly, and refer to its replica as $c'$. Since the formerly described algorithm only calculates a single

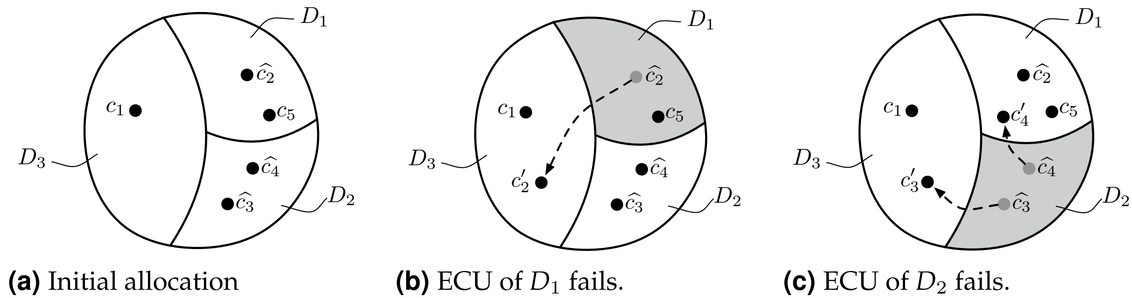
```
input  : COLA model of the Technical Architecture
output : Set of allocations 𝓐

1  𝓐 ← ∅
2  A ← InitialAllocation()
3  𝓐 ← 𝓐 ∪ {A}
4  foreach D_i ∈ A do
5      R = {ĉ ∈ D_i | ĉ is a marked cluster}
6      if |R| > 0 then
7          B_i ← D_i \ R
8          A' ← (A \ {D_i}) ∪ {B_i}
9          B ← ReplicaAllocation(A', B_i, R)
10         𝓐 ← 𝓐 ∪ {B}
11 return 𝓐
12 function Allocation ReplicaAllocation(Allocation A', Set B_i, Set R)
13     foreach D_i ∈ A' do
14         foreach c ∈ D_i do
15             add(a_[c↦p_i] = 1)        // p_i ∈ P ECU associated w/ part D_i
16     foreach c ∈ R do
17         add(a_[c↦b_i] = 0)            // b_i ∈ P ECU ass. w/ part B_i
18         add(∑_{p∈(P\{b_i})} a_[c↦p] = 1)
19     return solve()
```

**Algorithm 9:** Redundant allocation of safety-critical clusters.

allocation, its adaptions are described in the following and depicted in Algorithm 9.

Starting with an initial allocation result $\mathcal{A}$ (cf. line 2), the algorithm checks for each part $D_i \in \mathcal{A}$, $1 \leq i \leq r$, whether there is a non-empty subset $R \subseteq D_i$ of marked clusters. If so, a partial allocation $\mathcal{A}'$ based on $\mathcal{A}$ is generated. It differs in that $D_i$ is replaced by $B_i$ (cf. line 8), which does not contain the marked clusters anymore, i. e., $R \not\subseteq B_i$. `ReplicaAllocation(`$\mathcal{A}', B_i, R$`)` takes the partial allocation $\mathcal{A}'$ and fixes already made assignments (cf. line 15) except that of $R$ whose elements are not allowed to be placed into $B_i$ (cf. line 17). Therefore, new constraints are added (using `add()`) to those used to compute the initial allocation (`InitialAllocation()`). The remaining elements of $R$ (cf. line 18) are then

allocated onto the other parts of $\mathcal{A}$ under optimisation considerations (using `solve()`), yielding a redundancy allocation $\mathcal{B}$. The set of all allocations $\boldsymbol{\mathcal{A}}$ is extended by $\mathcal{B}$. Figures 7.18(a) to (c) illustrates the presented algorithm.




**(a)** Initial allocation **(b)** ECU of $D_1$ fails. **(c)** ECU of $D_2$ fails.

**Figure 7.18.:** Figure (a) shows the initial allocation: $D_1 = \{\widehat{c_2}, c_5\}$, $D_2 = \{\widehat{c_3}, \widehat{c_4}\}$, and $D_3 = \{c_1\}$. Figures (b) and (c) show the replication of marked clusters: cluster $\widehat{c_2}$ and clusters $\widehat{c_3}$ and $\widehat{c_4}$, respectively, have to be reallocated. Unmarked clusters are fixed (cf. $c_5$ in (b)).

### Scheduling

The necessary changes towards fault tolerance concerning scheduling are of moderate effort. In contrast to the procedure described so far, multiple allocations have to be considered. The scheduling procedure thus has to be aware of sets of allocations, each of them having its own $\alpha$. Up to now, the number of generated schedule lists was only dependent on the number of operating modes. Thus the scheduling algorithm was executed once for a unique given allocation determined by the mapping function $\alpha$. Together with the allocation result all information about when, where, and what cluster to start can be derived. If at least one cluster of the allocation is marked as redundant, a new allocation is derived. Hence, Algorithm 8 has to be executed for each $\mathcal{A} \in \boldsymbol{\mathcal{A}}$ with the mapping function $\alpha$ in question.

The operating system's dispatcher is responsible to select the correct schedule with respect to operating mode and faulty hardware.

## 7.5. Related Work

During the last years, there has been a lot of work dealing with model-based development of embedded systems. However, when having a closer look, most of

them have drawbacks that are not consistent with the idea of a seamless modelling along of different levels of abstractions. Of course, the presented COLA automotive approach cannot compete with commercial tools in terms of tool qualification and field trial. However, the prototypic implementation comes up with many interesting features that are not available in other approaches in such a seamlessly integrated form.

The AutomotiveArchitect [113], for example, describes an automotive architecture optimisation approach. The idea is to allocate functional components integrated in a functions net onto a hardware platform. Kebemou and Schieferdecker, however, do not model the behaviour of those components, which would be essential for a realistic automated deployment using optimisation and code generation. Moreover it is unclear how details of the hardware capabilities and the software resources are modelled. Lacking behavioural modelling capabilities is something that many other approaches have in common. Kebemou and Schieferdecker [114] also criticise that high level modelling languages such as SysML, EAST ADL, AADL, etc. are not expressive enough to support an automated partitioning approach for automotive systems. Detailed information about the underlying hardware topology, their attributes and a behavioural description is needed.

Lately UML has become popular for modelling real-time systems. One approach to generate C code from UML models has been presented by Khan et al. [115]. But compared to COLA, the current diagram types defined in UML do not provide enough information for generating the entire application code, thus having the need to manually write code. Avoiding those error prone manual changes to the resulting code was one of the main reasons for using a data-flow language like COLA in the current work. The information captured in a COLA model is sufficient to generate code necessary for a runnable system. The UML profile MARTE (**M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded Systems) [155] is currently in the course of standards definition. Therefore, Espinoza et al. proposed an annotation of UML models with non-functional properties [68]. UML, however, is a general-purpose language, which does not cater for the specific needs of the sub-domains of embedded systems design, like automotive or avionics industry. In [28], Broy objects and favours the use of domain specific languages and architectures to improve the state-of-the-art. Therefore, we use COLA as such a domain specific language, which, in contrast to UML, also features a unique formal semantics.

Considering model-based engineering of embedded control software, Schätz proposes in [178] a clear separation of control- and data-flow models to avoid un-

necessary complexity. Control-flow is used to specify modes of operation, whereas data-flow is used to define the mode's control task. In a similar way, COLA models are structured with respect to operating modes using mode automata. As an essential improvement—especially in the context of safety-critical systems—the COLA automotive approach describes a novel technique to generate executable code for complete systems, where operating modes are distributed over several computing nodes.

AUTOFOCUS is a further academic tool, which is currently in the process of being used in an industrial setting. Similar to the presented approach, it supports seamless model-based development along different levels of abstraction. In contrast to the COLA-IDE, AUTOFOCUS is based on the FOCUS theory. Both approaches are very comparable in terms of functionality. However, they differ in advanced features:

(i) the COLA-IDE supports the generation of requirements specification documents (cf. Chapter 6),

(ii) it provides support for model-based debugging (cf. Herrmannsdörfer et al. [96]),

(iii) moreover a traceability relation can be visualised between the Feature and the Logical Architecture using 'chain of effects' (cf. Section 8.3.3), and

(iv) concerning deployment, COLA does support separate deployment of operating modes, i. e., states of mode automata, which helps to save valuable resources.

On the credit side of the account, AUTOFOCUS enters a long-time development and evaluation also in other domains besides the automotive one, e. g. plant automation, avionics, and energy engineering.

Common commercial tools like MATLAB/Simulink [188] and ASCET-SD [25] are used for the design of embedded systems. These tools also feature automatic code generation as described by Putty et al. [169,200], but the employed generators are limited to translate the functionality of the modelled system. Adding system functionality like for instance fault tolerance code is beyond their scope. This is partly caused by the fact that COLA is aimed at the design and synthesis of distributed systems, while most commercial tools are focusing on non-distributed per-ECU modelling.

For SCADE [3], which is based on Lustre [91], a deployment concept for distributed embedded systems has been presented by Caspi et al. [42]. Compared

to COLA, this approach lacks a key concept: it does not offer the automatic deployment of operating modes, supporting a dynamic change of schedule plans at run-time, although Lustre and SCADE support mode automata [132, 146]. Furthermore, unlike the COLA modelling process, no optimised automatic allocation is performed.

Another language with similar focus regarding the target systems is Giotto, as presented by Henzinger [94]. Models specified therewith are also executed in a cyclic manner, similar to COLA models. An extension of Giotto towards distributed platforms has been described in [95], namely Distributed Giotto. Unlike COLA, Giotto defines the causal order of tasks and their resource requirements, but does not deal with specifying their implementation. Rather tasks are implemented by hand and the Giotto compiler guarantees the timely execution in a distributed system, given worst-case execution times and call frequencies for all tasks are known. COLA contrariwise defines the clusters' implementation, which is necessary for formal verification and calculation of reliable execution times based on the designed model, as presented by Wang et al [194].

Annotation of non-functional requirements and a notation of platform capabilities were described by Dinkel and Baumgarten [60]. Their goal, however, was the dynamic system reconfiguration at run-time. Not only non-functional requirements and capabilities are modelled, but describe a fully automatic deployment process. Wuyts and Ducasse [199] instrument components, with non-functional requirements, specified in Comes (a general Component Meta-Model). In Comes, components are seen as black boxes annotated with properties. This may be sufficient for allocation and scheduling, but lacks the information necessary for model checking and other verification techniques. In COLA, each level of abstraction—from a very high-level system design down to the low implementation level—offers a white-box view and therefore provides all necessary information.

Moreover, Matic et al. [148] take platform specifications, e. g. power modes of the micro-controller, into account as well as application specific information like periods of tasks in order to generate an optimal scheduling. Compared to the presented approach, their work starts from having tasks to schedule. The approach of this thesis, however, supports an integrated development process of distributed hard real-time systems from requirements engineering (system features) over the design phase to the actual code generation, task allocation, and scheduling in a consistent modelling formalism. Furthermore, an objective function is optimised subject to constraints stemming from non-functional requirements.

Regarding the overall design process, the DECOS project [121] is close to the pre-

sented approach. Unlike COLA, however, they do not use a consistent modelling formalism, but rather resort to various techniques.

Support for automatic integration of fault tolerance into safety-critical systems is very desirable in many cases. As a study by Mackall and colleagues showed [142], all failures observed in the reviewed system where due to bugs in the design of the fault tolerance mechanisms themselves. And as Rushby showed in his overview of fault-tolerant bus architectures [175], even up-to-date bus systems like FlexRay not necessarily have integrated fault tolerance mechanisms. The concept presented in this work thus would be a valuable complement for those systems.

**Time-Triggered Architecture (TTA)**

The requirements given for the used platform concept are similar to those of the TTA, the *Time-Triggered Architecture* [116, 118, 119] presented by Kopetz et al. In contrast to the TTA, the used middleware allows, in theory, an arbitrary number of operating modes, while the TTA is limited to eight. This is advantageous as the use of operating modes for fault tolerance presented here further raises the number of possible modes. Contrariwise, COLA automotive approach lacks the hardware support for failure detection by a bus guardian as intended for the TTA. The TTA communication system periodically executes a time-division multiple access (TDMA) schedule. Access to the communication medium is divided into a series of intervals, so-called *slots*. The exact times, where a node is allowed to send messages over the communication medium is calculated *a priori*, i. e., at design time (cf. also [85]).

**Time-Division Multiple Access (TDMA)**

The COLA automotive methodology focuses on an *optimised* automatic deployment process for embedded hard real-time systems with respect to a set of given non-functional requirements. These requirements are considered by the presented allocation algorithm. Similar to Zheng et al. [205] and Matic et al. [148] integer linear programming (ILP) is used. In addition to Zheng et al., Metzner and Herde [150] who are using a SAT-based approach for the task allocation problem, the presented approach takes non-functional requirements during the deployment process into account (cf. also Section 7.2.2). De Niz and Feiler [53] present a resource allocation approach using the OSATE tool for AADL architectural models. Their approach uses the Partition Bin Packing algorithm, which is an extension of the **B**est-**F**it **D**ecreasing (BFD) bin backing algorithm to bind entities of the software model to entities of the hardware model. Considering the change of operating modes of hard real-time systems, Fohler [76] proposes a pre run-time solution for the MARS approach [120].

## 7.6. Summary

In this chapter, the deployment concept realised within the COLA automotive approach was discussed. Following the IPO-model during the deployment steps, the semantics of COLA's synchronous data-flow language is preserved down to the execution platform. This distinguished characteristic addresses and solves problem **P4** posed by Sangiovanni-Vincentelli and Di Natale [177] and took up in Section 3.1.

The allocation step uses an optimisation scheme primarily based on non-functional requirements to compute an optimal—according to the desired optimisation goal—allocation, i. e., assignment of clusters to hardware entities. The approach uses an encoding into an integer linear program to search for an optimal partitioning of the E/E architecture.

The presented scheduling procedure as it is used within the COLA automotive approach uses SMT solvers to find solutions for the complete system also with respect to different operating modes. Each such solution satisfies the mentioned constraint system. In order to find a solution, which is also optimal in a sense that it minimises the make span, binary search is used. The scheduling analysis and computation address Sangiovanni-Vincentelli's and Di Natale's problem **P3** mentioned in Section 3.1. The regarded case studies have in common that all clusters have the same periods. An extension to multi-rate systems is given, too.

Figure 7.19 depicts the deployment perspective of the graphical user interface of the COLA-IDE.

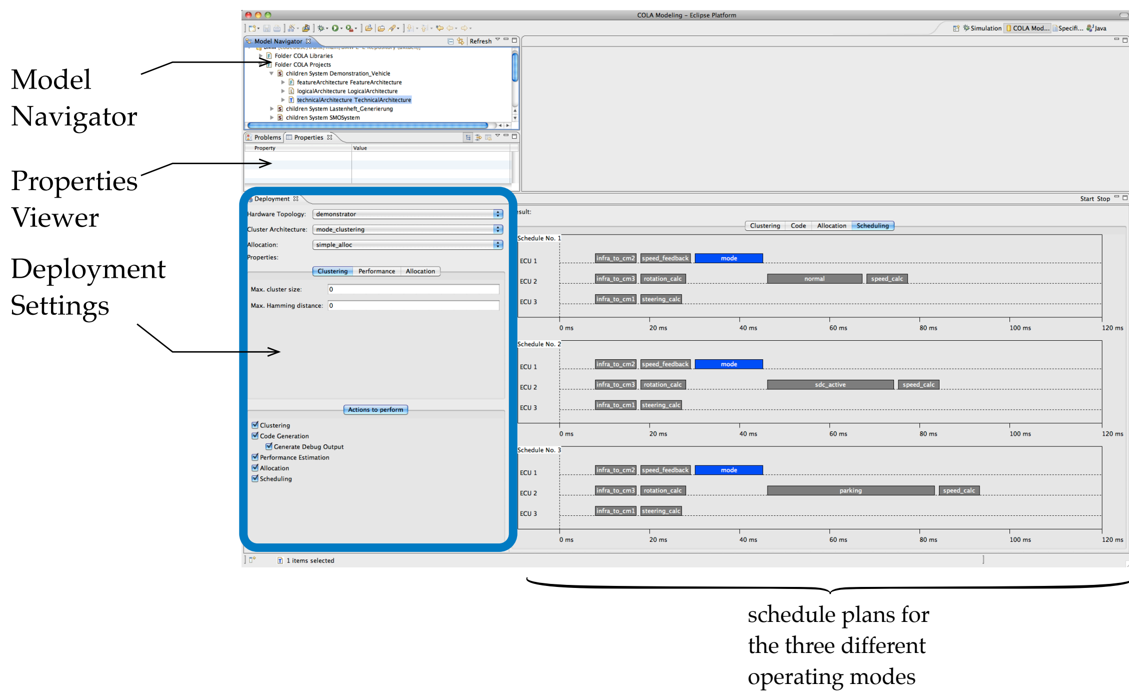



**Figure 7.19.:** COLA-IDE deployment perspective.

CHAPTER

# EIGHT

## Case Studies

In this chapter, case studies will be given that show the feasibility of the COLA automotive approach. Each of the examples pursues a different goal. As a consequence, not all capabilities of the COLA-IDE are used in every example but rather particular capabilities. The first case study is an adaptive cruise control system (ACC) discussed next in Section 8.1. An autonomous parking system (APS) has been realised as second case study described in Section 8.2. Focus of the third case study was to show that it is indeed possible to develop a totally new function—an comfort hatchback opener (CHO)—very fast (cf. Section 8.3).

There are several reasons why not all capabilities of the COLA-IDE were used in every case study: first and foremost, not all capabilities had been developed and integrated in the COLA-IDE when the first case study was realised. Second, priorities were set by the industrial collaborator. Figure 8.1 visualises the focus of each case studies.

### Contents


8.1. Adaptive Cruise Control . . . . . . . . . . . . . . . . . . . . . . . . 182

8.2. Autonomous Parking System . . . . . . . . . . . . . . . . . . . . . . 185

8.3. Comfort Hatchback Opener . . . . . . . . . . . . . . . . . . . . . . . . 191

8.4. Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 195

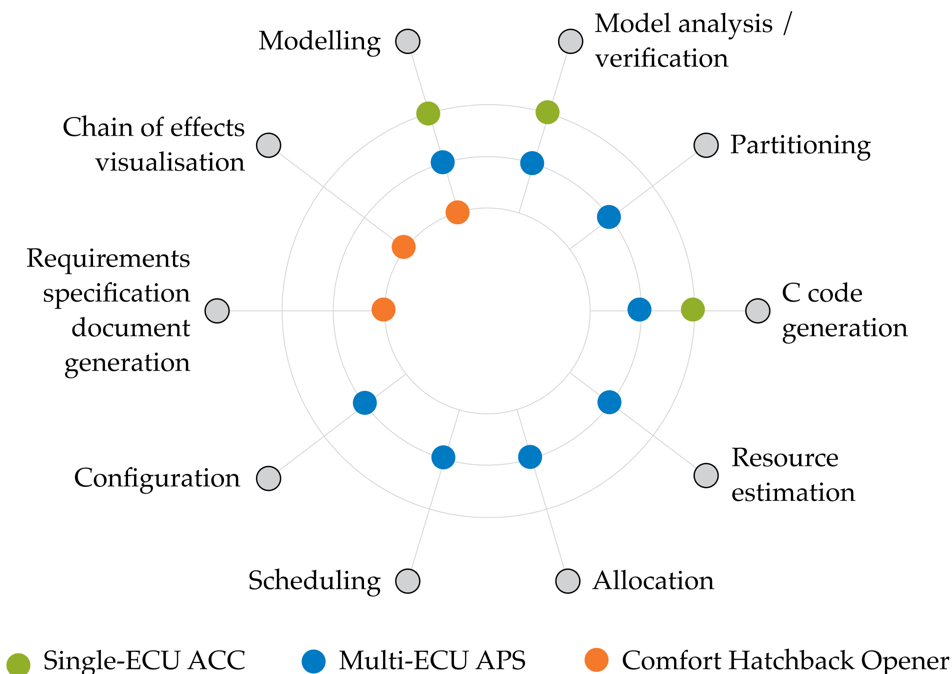



**Figure 8.1.:** Focused activities of the case studies.

## 8.1. Adaptive Cruise Control

The focus of the adaptive cruise control case study was to show the modelling capabilities of the COLA automotive approach. Moreover, the generation of C source code was used first. I want to thank Michael Tautschnig and Wolfgang Haberl who modelled the functional behaviour as well as built up the execution platform.

**adaptive cruise control**

The ACC system is an advanced driver assistance system. In addition to a standard cruise control system, which automatically controls the speed of a motor vehicle in order to maintain a steady speed as set by the driver, the ACC system also records the distance and the relative speed of a vehicle driving ahead. These data are used to control the distance between both vehicles. The following Section 8.1.1 gives a functional description followed by information about the execution platform in Section 8.1.2. Section 8.1.3 gives a summary. As that was the first case study, the proof-of-concept and not a complete and comprehensive functional and technical perfect realisation had priority.

### 8.1.1. Functional Description

As a first prototype, intended to demonstrate the proof-of-concept, the implemented ACC functionality is limited to follow a vehicle driving directly in front. The driving behaviour is limited to go straight on without driving curves. The complex interaction of the ACC module in real-life vehicles, i. e., interaction with electronic stability control system, the gearbox, and the engine control—just to mention the most important—is reduced to the most necessary: an engine, a distance measure sensor, and a rotation sensor. Therefore, the functional range is less than its example on the road. Since the demonstrator is intended to move straight ahead, there is no need to determine the route of the ahead-driving car in winding roads or even to determine its traffic lane. As a consequence, the functional behaviour could be realised without information like yaw rate, steer angle, and lateral acceleration. The following list contains the complete scope of operation of the demonstrator:

(i) The system can be activated and deactivated using the buttons on the LEGO brick.

(ii) The desired speed can be adjusted using the buttons.

(iii) If the system is deactivated, the currently set speed is sent to the engine, moreover the display shows 'off'. If the system is activated and the distance to the ahead driving vehicle is more than $35$ the display shows the speed set by the user. Furthermore, if the system is activated and the distance is less than $35$, the display shows 'slow'. Is the distance zero, the system indicates this circumstance with 'stop' in the display.

(iv) If the system is activated, the set speed is controlled such that a steady speed is maintained also in case of changing pavement or incline conditions.

(v) As long as an obstacle is detected within the vehicle trajectory within a maximum distance of $35$, the system continuously reduces the speed by 5%. Below a threshold of $15$ (minimum distance), the system performs an emergency stop.

(vi) As soon as there are no more obstacles within the vehicle's trajectory, the speed increases stepwise by 5% of the difference between the current and the desired speed. This ensures a smooth acceleration until the desired speed has been reached, or an obstacle is again in the vehicle's trajectory.

The smooth acceleration and breaking behaviour is illustrated in the following example.

**Example.** Assume the ACC-powered vehicle follows another vehicle driving at a constant speed of 20 ahead starting with a distance of 10. The ACC vehicle starts with an initial speed of 5, however, the desired speed is set to be 25. In Figure 8.2, a simulation with the stated values is depicted. The green curve indicates the distance between both vehicles over time. The figure shows the simulation for 250 steps. The orange line indicates the constant speed of the ahead-driving vehicle, whereas the green one illustrates that of the ACC-powered vehicle.

Since the initial distance of 10 is less than the minimum distance of 15, the vehicle does not move at all as it performs an emergency stop. Only when the obstacle moves on and thus increases the distance, the following car also starts to move. As long as the obstacle's speed is greater than the vehicle's speed, the distance between both vehicles increases more and more. But then, when the vehicle's speed exceeds 20, the distance begins to shrink—in this example after 32 steps. This phase ends just as the distance is below 35, that value when the vehicle begins to break. After about 130 steps, the distance levels off at about 35 and the speed at around 20, which is the speed of the obstacle driving ahead.

### 8.1.2. Hardware Topology and Execution Platform

**LEGO Mindstorms**

The hardware topology is mainly determined by the chosen LEGO Mindstorms execution platform. The LEGO brick offers two softkeys used to adjust the desired speed (increase/decrease). The ACC can be activated and deactivated with a touch sensor used as a button. The ultrasonic and rotation sensors are connected to the sensor ports, whereas the electromotor and the break lights are connected to the actuator ports of the RCX. The display is used to give feedback of the vehicle's current operating mode and the set speed, respectively. Figure 8.3 gives an overview of the hardware topology. The capabilities of the LEGO Mindstorms platform are very restricted and thus can be considered as realistic for embedded systems. The programmable LEGO brick RCX is powered by an 8-bit Renesas (formerly Hitachi) H8/300 microcomputer. The 32K of RAM are used to store the firmware as well as user programs. It offers a display for short messages, two softkeys, and ports to connect sensors and actuators—three each. BrickOS was used as operating system.

**BrickOS**

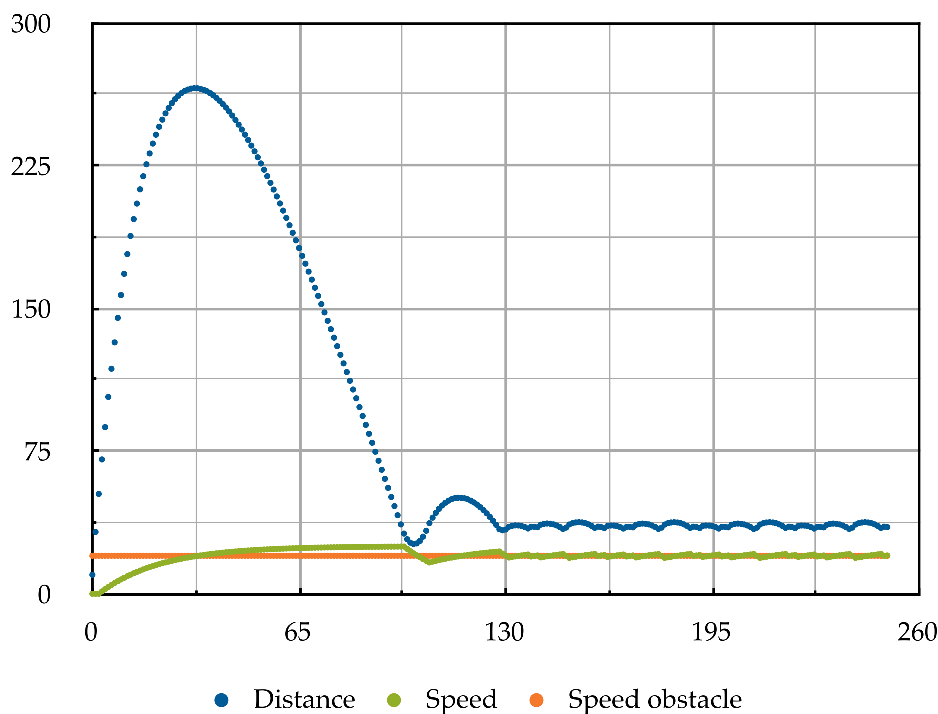



**Figure 8.2.:** Simulation of the ACC behaviour (250 steps).

### 8.1.3. Summary

The aim of this first case study was to demonstrate the feasibility of the presented MDD approach. As the capabilities of the used demonstration hardware platform were very limited, a distributed system was not realised. As the functionality has been realised as one monolithic task, no scheduling has to be done. But, this demonstration vehicle showed very well the transition from a COLA model to an executable C code generated by the developed code generator [90]. Figure 8.4 shows the LEGO vehicle.

## 8.2. Autonomous Parking System

The second mayor case study was a multi-functional, distributed, network embedded system, namely an **A**utonomous **P**arking **S**ystem (APS), which is a non-trivial function available in modern medium- and luxury class cars. The major difference between the ACC example and this one is that the full deployment capabilities of the COLA-IDE, namely model partitioning, allocation, scheduling, C code

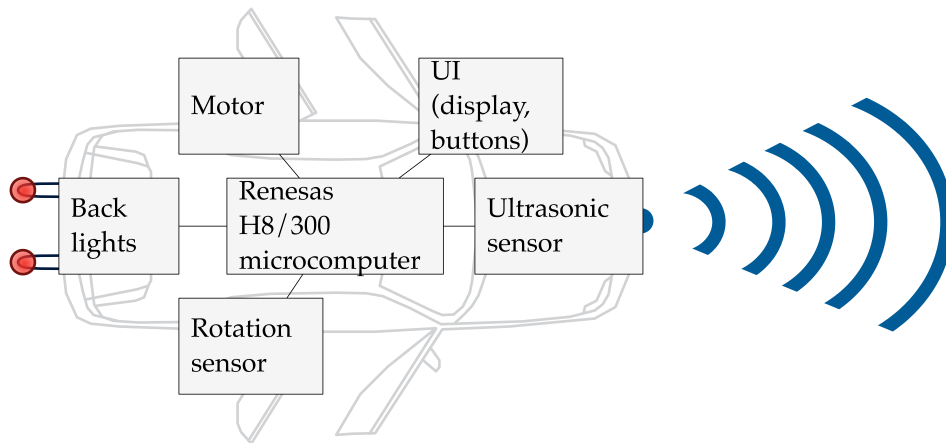



**Figure 8.3.:** Hardware topology of the LEGO Mindstorms ACC case study.

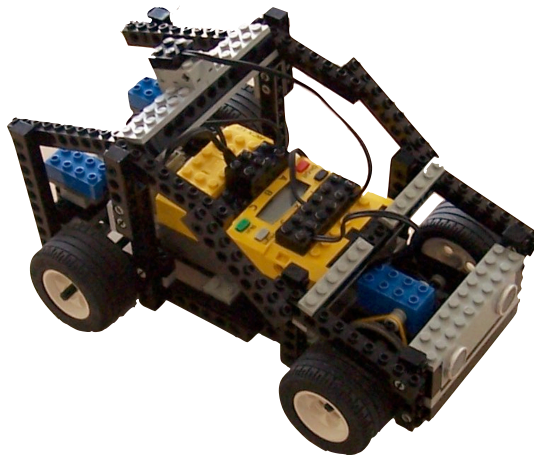


**Figure 8.4.:** LEGO Mindstorms ACC.

generation, and platform configuration were applied. Although the parking capability is at the centre of attention, the demonstration car on a scale 1:10 also allows manual steering via Bluetooth and a side clearance control. Hence, we call it a multi-functional system even though with a quite limited functional range. The

key features of this system are:

(i) Finding and measuring of a parking space and

(ii) automatic reverse parking without

(iii) bumping any obstacles.

The following sections will detail on the functional description in Section 8.2.1, the hardware topology in Section 8.2.2, and the execution platform in Section 8.2.3. Finally, this case study is concluded in Section 8.2.4.

## 8.2.1. Functional Description

Besides the parking functionality, the system is also able to be steered manually via Bluetooth connections to a customary mobile phone. Furthermore, the car can follow a wall keeping a constant distance and perform an emergency stop if an obstacle appears in front of the car that cannot be circumnavigated. The basic functionalities are modelled using the concept of operating modes. Therefore, each of the modes 'normal', 'sdc_active', and 'parking' (cf. Figure 8.5) is encapsulated in one state of the depicted automaton 'vehicle_mode'. In the mode 'normal' manual

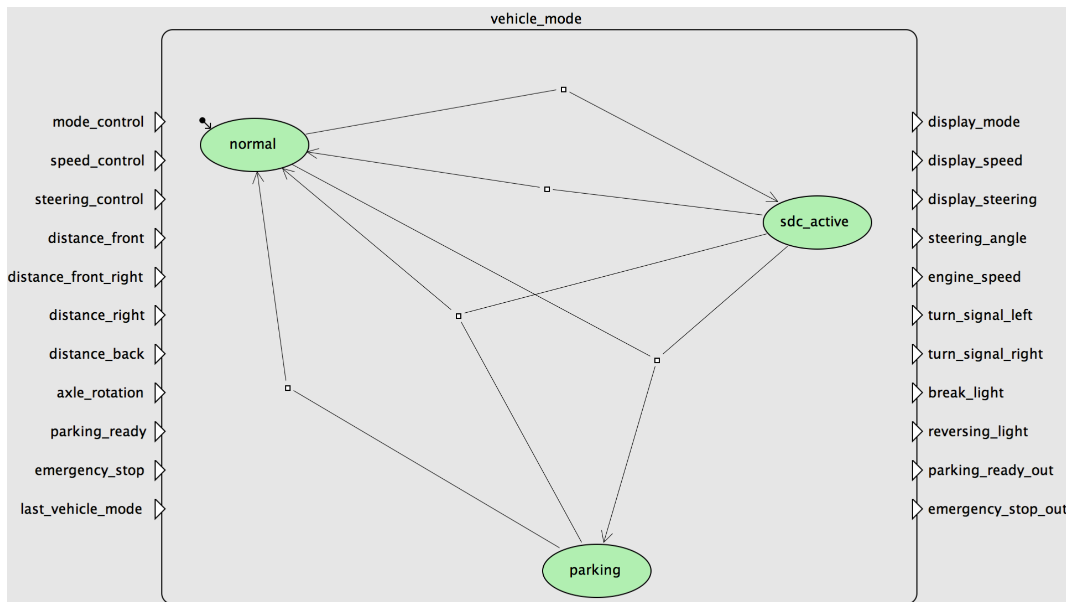



**Figure 8.5.:** Operating modes of the multi-ECU case study modelled on the Logical Architecture.

steering via Bluetooth connections to a mobile phone is possible, 'sdc_active'

describes the mode of following a wall at a predefined distance without hitting any obstacles, and last but not least 'parking' encapsulates the parking behaviour. This method, again, facilitates a separation of concerns, decoupling of different functions, and—when thinking about the runtime behaviour defined on level of the Cluster Architecture—reduces the load of a certain ECU, since only those clusters dedicated to a certain operating mode have to be executed at the same time. Figure 8.6 depicts in more detail the parking feature. It basically consists

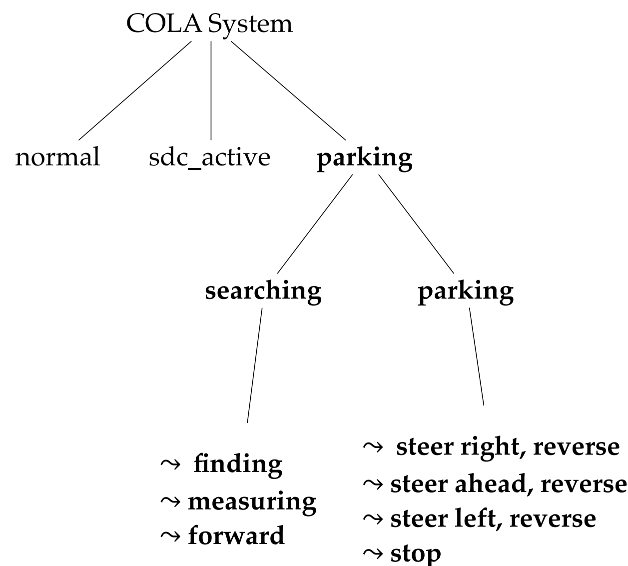



**Figure 8.6.:** APS functionality in detail.

of the two steps 'searching', i. e., the identification of a parking slot of sufficient size, and the step 'parking' that performs the automatic reverse parking. The search steps subsume to find a parking slot and to measure its size. If the size is sufficient, the car drives a little bit forward before initialising the parking activity. The parking algorithm works similar to how one is used to do reverse parking: first, one reverses the car while steering right, followed by reversing while steering ahead in the second step. Next, in the third step, the steering is changed to left until the final parking position has been reached.

All the performed steps can be visualised using the COLA simulator developed by Herrmannsdoerder at al. [96] (cf. Figure 8.7).

### 8.2.2. Hardware Topology

The graphical model of the Hardware Topology as modelled in the COLA specification language can be seen in Figure 8.8. It consists of the three micro-controllers

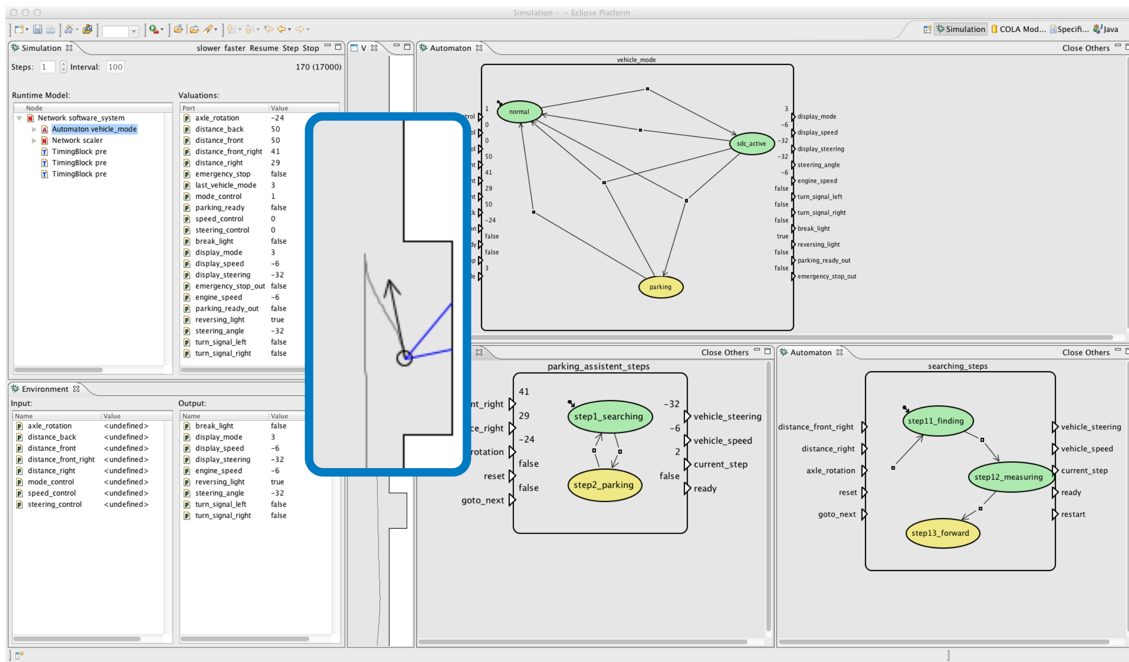



**Figure 8.7.:** COLA simulator [96] used to visually simulate the parking functionality.

'CONNEX_1' to 'CONNEX_3' connected via a common bus 'CONNECTION_1'. Each of the micro-controllers (ECU) comprises of a connection interface, a CPU and a couple of directly connected sensors and actuators. Details of the Hardware Topology and its entities are listed in Table 8.1. Each of them represents a con-

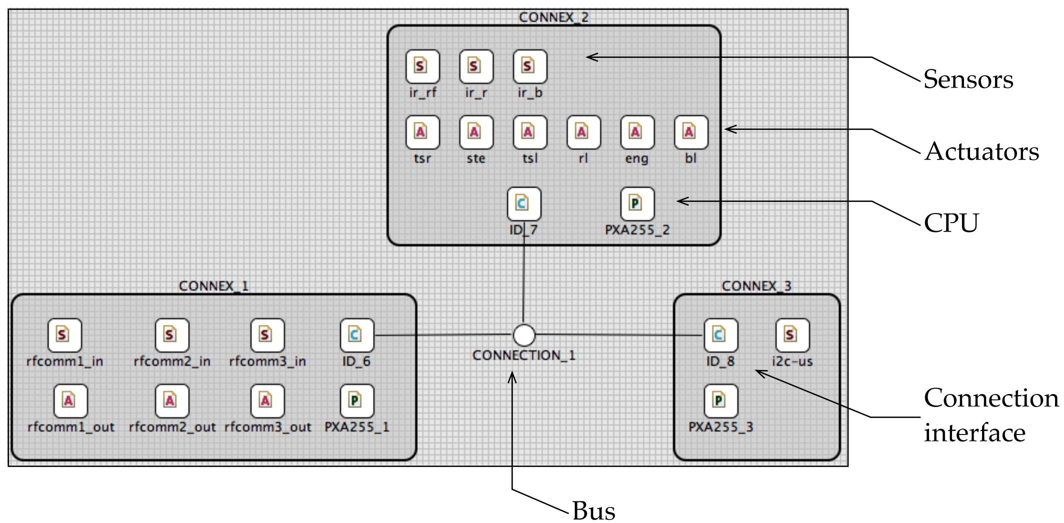



**Figure 8.8.:** Hardware Topology of the APS case study.

crete physical device in the execution platform discussed next. As one can see, ECU 'CONNEX_1' is responsible for data communication to the mobile phone, 'CONNEX_2' takes the part of triggering the actuators except that for the display, which is done by ECU 'CONNEX_3'.

| Component | Description |
|---|---|
| rfcomm1_in, rfcomm2_in, rfcomm3_in | Incoming Bluetooth connections from the mobile phone. |
| rfcomm1_out, rfcomm2_out, rfcomm3_out | Outgoing Bluetooth connections to the mobile phone. |
| CONNEX_1, CONNEX_2, CONNEX_3 | Intel XScale PXA255 micro-controllers running at 400 MHz with 64 MB of RAM and 16 MB flash memory. |
| ID_6, ID_7, ID_8 | Connection interfaces. |
| ir_rf, | Infrared distance sensor assembled on the front-right bumper. |
| ir_b, | Infrared distance sensor assembled on the rear bumper. |
| ir_r, | Infrared distance sensor assembled on the right door sill. |
| bl | Actuator for the break light. |
| eng | Motor actuator. |
| rl | Actuator for the reversing light. |
| ste | Steering actuator (servo) |
| tsl | Actuator for the left indicator. |
| tsr | Actuator for the right indicator. |
| i2c-us | $i^2c$ bus display interface. |
| CONNECTION_1 | Ethernet connection. |

**Table 8.1.:** Components used to build the Hardware Topology.

### 8.2.3. Execution Platform

The execution platform can roughly be described by mechanical and E/E components, which in turn may either be material or immaterial. Purely mechanical components of that case study is for instance the assembly set of bodywork and chassis. ECUs, Ethernet-switch, engine etc. fall into the category of material E/E components. Immaterial is—of course—the deployed software and the operating system, which is a Linux with the Xenomai real-time development framework.

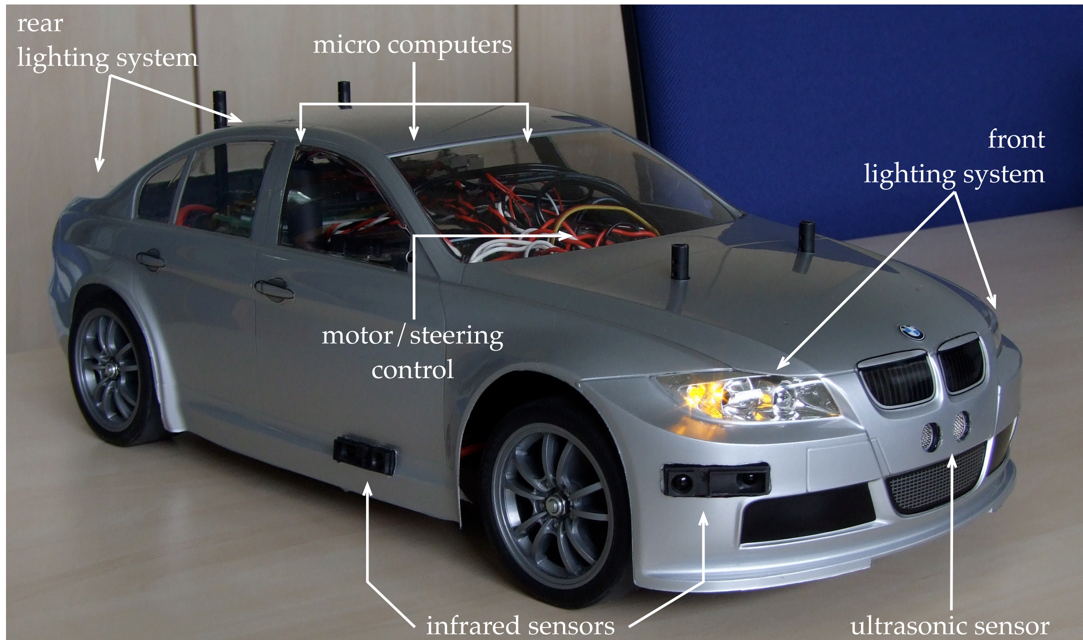
Communication between ECUs is done using Ethernet and a middleware developed by Haberl et al. [85]. As ECUs, three Gumstix 400xm-bt boards with Intel XScale PXA255 processors, 64 MB RAM, and 16 MB of flash memory were used. These boards were extended with a netCF Ethernet adapter each. Two infrared and an ultrasonic sensor were used for distance measurement, an electric engine as propulsion, a steering servo, and a couple of lights (stop lights, direction indicators). A Nokia 6630 was used to steer the car and to set the operating modes via Bluetooth. In Figure 8.9a a photo of the case study is depicted. At this point, I want to thank my workmate Wolfgang Haberl who built up the hardware of this case study in charge with a lot of passion and effort. Figure 8.9b shows the visualisation of the three scheduling plans, one for each operating mode.

### 8.2.4. Summary

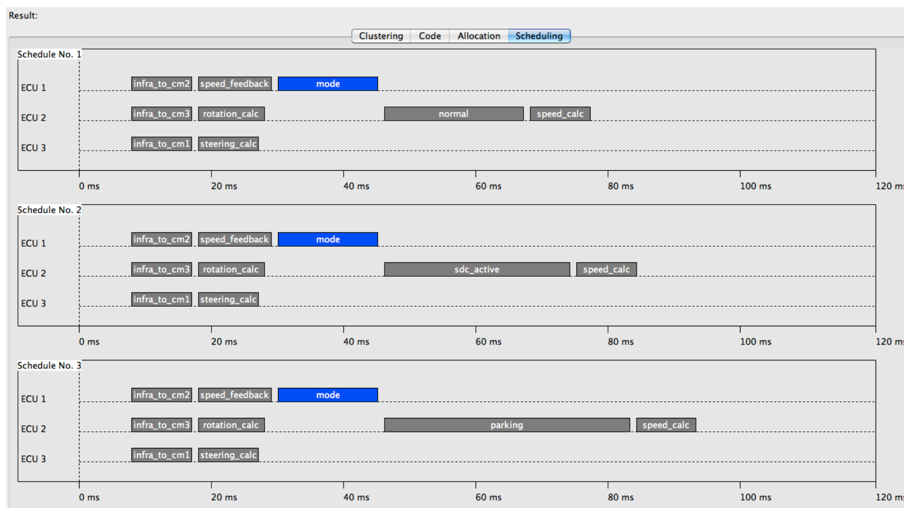
The intention of the presented case study was first and foremost the extension from a single-ECU system to a real distributed multi-ECU networked embedded system. Therefore, a deployment concept for COLA systems had to be developed including allocation (cf. Section 7.2) and scheduling (cf. Section 7.3) with respect to different operating modes. This case study used the capabilities of the COLA-IDE on each level of abstraction. Hence, a fully integrated COLA engineering tool supporting seamless model-driven development could be presented the first time [86, 87].

## 8.3. Comfort Hatchback Opener

The case study described in this section has been modelled based on a real requirements specification document of BMW Group. The COLA model has been developed in parallel to the realisation of the first tier supplier, who developed the system for Audi, Volkswagen, and BMW Group. Due to non disclosure agreements, not all realisation details can be given. Besides a feature model on the Feature Architecture, a simulatable behavioural model has been developed. In addition, this model formed the basis for the generation of a requirements specification document for that particular function. One way to visualise the connection between the Feature and the Logical Architecture is the so-called 'chain of effects' which has been demonstrated using the **C**omfort **H**atchback **O**pener (CHO) example. The deployment onto a real car was not possible at that time.




(a)




(b)

**Figure 8.9.:** (a) Image of the APS case study with (b) the corresponding schedule plans.

## 8.3.1. Functional Description

The idea of the comfort hatchback opener is to enable the driver to guesture-driven open the hatchback of his or her car. This is a very useful feature especially after shopping when loading the trunk with full hands. The challenge of this feature is not to model the desired 'good' case, but the undesired 'bad' cases, i. e., the

situations when the system should not open the hatchback.

After detecting a correct gesture, the system has to perform several checks before opening the hatchback. These are amongst others to check if sufficient battery voltage is available, if no hardware malfunction was detected, and if the correct key is present in a certain area relative to the car. This example demonstrates, that in many cases, the modelling effort for the 'good' case is less compared to the 'bad' cases. This becomes apparent when having a look at the anonymised Feature Architecture given in Figure 8.10. Some information are camouflaged such that no information about the real requirements specification documents can be gained. Actually, the desired opening functionality is encapsulated within the

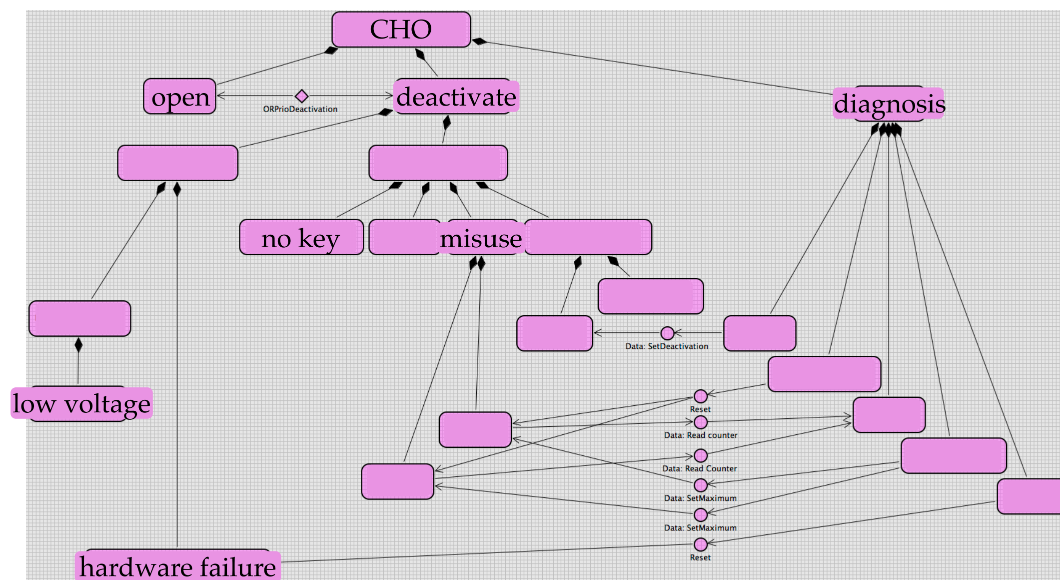



**Figure 8.10.:** Anonymised Feature Architecture of the CHO example.

feature 'open', whereas 'deactivate' and 'diagnosis' with their respective subtrees contain actions to be performed if for example no key is present or misuse has been detected. Moreover, a lot of diagnosis functionalities have to be realised apart from the open functionality. If all mentioned checks are passed and a correct gesture has been detected, the hatchback opens. Essentially, the algorithm compares the voltage curve given by sensors with a predefined curve. Due to non disclosure agreements, however, no more information about how the detection is technically realised and how the detection algorithm works can be given. I want to thank Doris Wild and Bernd Spanfelner for modelling the Feature and the Logical Architecture of this example.

### 8.3.2. Requirements Specification Documents

Besides the proof of concept that the COLA automotive approach can be used to develop simulatable models even for new functions very quickly, this case study served as source model to generate requirements specification documents as discussed in Chapter 6 based on the Logical Architecture.

### 8.3.3. Chain of Effects

**chain of effects**

**impact analysis**

Requirements traceability is a mandatory property during system development demanded for instance by ISO 26262. The notion of a *chain of effects* is one way to graphically visualise how features of the Feature Architecture are realised on level of the Logical Architecture. That means, they visualise the *impact* on the logical model when changing a feature. Of course, this makes no claims of being complete, however it might be sufficient in many cases. Realistically, one has to mention, that for automotive systems of realistic size, it will never be possible to have 100% complete traceability. This might be because of incomplete information or insufficient tool support. Analysis also showed that automotive systems are interconnected in such a way that nearly all components are related to certain requirements. This makes it de-facto unusable for engineers.

The fashion how the chain of effects are implemented in the COLA-IDE is referred to as *post-traceability*, as it provides insights *after* the Feature and the Logical Architecture have been created. In principle, the COLA engineering environment could support full traceability when unifying the following information:

(i) To feature annotated requirements,

(ii) chain of effects to explain the interconnection between Feature and Logical Architecture,

(iii) the Cluster Architecture, to know how logical components are clustered, and

(iv) the way how the COLA C code generator translates clusters into source code.

All this information could provide detailed insight of how a feature is logically modelled and technically realised in code. Figure 8.11 depicts an exemplary chain of effects for the feature 'no key'. The algorithm building this chain of effects simply matches the selected features input and output ports, and locates them on the Logical Architecture. By performing a

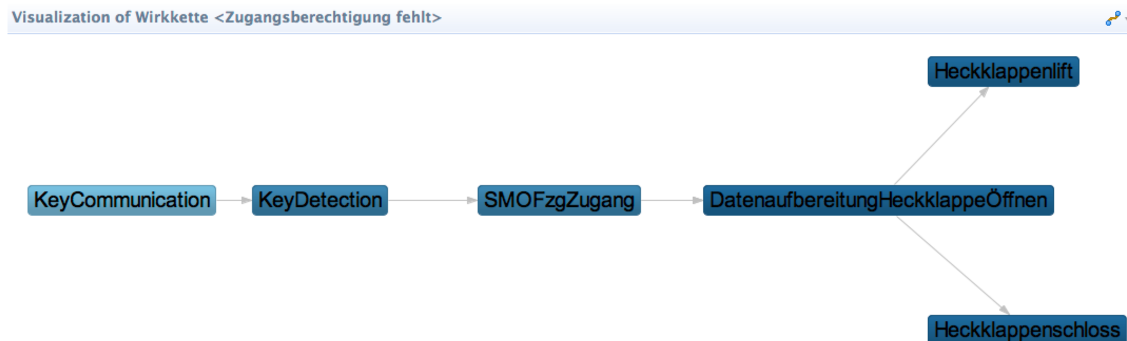



**Figure 8.11.:** Chain of effects for the feature 'no key'.

(i) backward search from the input ports of the corresponding unit on the Logical Architecture, via connected channels, until sources have been reached and a

(ii) forward search from the output ports of that unit until sinks have been reached.

By collecting all units visited during the two search phases, the chain of effects is constructed. For reasons of clarity and comprehensibility, in a first step not all details are shown. If desired, however, more details can be shown, i. e., a node for example 'SMOFzgZugang' will be replaced by its contained subtree. The herefrom gained hierarchy is indicated using different node shapes—the lower in the hierarchy the darker the shape.

### 8.3.4. Summary

The last case study pursed the goal to demonstrate that completely new functions can be modelled quite quickly. Besides that, a semantically tangible requirements specification document was created and the correlation between the Feature Architecture and the Logical Architecture has been shown using the notion of a chain of effects.

## 8.4. Summary

This chapter demonstrated the feasibility of the COLA automotive approach by dint of three different case studies. Each of them focused different aspects during development:

(i) The adoptive cruise control system demonstrated the practicability of C code generation.

(ii) The autonomous parking system revealed a seamless modelling along different levels of abstraction using a distributed, multi-functional system.

(iii) The comfort hatchback opener focused on the generation of requirement specification documents as well as the visualisation of the correlation between the Feature and the Logical Architecture using chains of effects. Moreover, one could demonstrate how fast simulatable models for completely new functions can be modelled.

All aspects together claim to cover the central ideas of what was proposed as 'seamless model-based development' of software-intensive automotive systems.

CHAPTER

# NINE

# Summary and Outlook

This concluding chapter summarises the central results and contributions of this thesis and discusses further possible research directions. In the following section a summary is given followed by Section 9.2, which reveals possible future areas of research.

**Contents**


9.1. Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 197

9.2. Outlook . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 199


## 9.1. Summary

This dissertation proposed a novel approach for seamless model-based development of reliable software-intensive embedded automotive systems. The particular requirements, problems, and challenges of the automotive domain were given in the beginning, namely the highly competitive mass market, the omnipresent dependency on suppliers, legal requirements, and the drift to hybrid powertrains and electromobility, which is a challenge and an opportunity at the same time. To the best of our knowledge, there is no tool—let it be commercial or from the academia—that comprises a similar high and especially seamlessly integrated functional range building upon a common product data model.

After an overview of embedded systems had been given and their penetration into automobiles during basically the last 40 years had been recapitulated, the automotive industry was elucidated in terms of its domains infotainment, body electronics, chassis, powertrain, and passive safety.

The COLA automotive approach supports modelling along different levels of abstraction—the Feature Architecture, the Logical Architecture, and the Technical Architecture. Dijkstra's [58] and Parnas' [157] ideas to structure and modularise systems are not only supported on all levels of abstraction, rather these were specifically designed to do so. The COLA automotive approach is characterised by the fact that the core of the used modelling language COLA—The Component Language—is mathematically well-defined with respect to its formal syntax (textual as well as graphical) and semantics. In fact, this rigorous foundation is at the very heart of the presented work. This is crucial for the application of formal methods such as model checking, simulation, or other verification techniques. It is this fact that enables the application of model-to-model transformation like that presented in the case of Coloured Petri nets, and synthesising of code. Model-checking techniques were used to demonstrate how inconsistencies among a set of features can be detected. In addition, a method was presented to check whether a COLA automaton is deterministic or not when it is considered individually. The quoted algorithm utilises and benefits from the performance of state-of-the-art SMT solvers.

A per-tool formal foundation in a chain of different tools, however, undermines the idea of seamless modelling by encouraging integration gaps, which may lead to information loss or information corruption. On the contrary, the COLA-IDE integrates everything that is necessary to take the transition from a high level functional model down to very concrete executable code in a 'push-button' fashion. This dissertation emphasises within the involved deployment steps aspects concerning allocation, i. e., the mapping of logical software artefacts (clusters) onto processing units of the hardware model, and scheduling of distributed COLA systems. The first mentioned, moreover, takes non-functional requirements into account to optimise the allocation result.

It was shown, how requirements specification documents can be generated directly from the model of the Logical Architecture. So-called 'scope of services' define the scope of what should be part of the generated document. Due to the tight OEM-supplier relationship, parts that are intellectual property of the OEM will only be part of the document in a restricted form. The very tight integration of the document viewer into the COLA-IDE enables navigation through the model by clicking on elements of the document. This makes them 'semantically tangible'.

Finally, the three case studies

(i) Adaptive Cruise Control (ACC),

(ii) Autonomous Parking System (APS), and

(iii) Comfort Hatchback Opener (CHO)

showed the feasibility of the presented COLA automotive approach. Each of the examples focused different aspect, such that in their combination all capabilities of the COLA-IDE and the idea of seamless model-based development of software-intensive automotive systems could be demonstrated.

In particular, this dissertation contributed to the state-of-the-art and state of practice by

(i) proposing a seamless model-based development approach for primary automotive systems, which uses the COLA modelling language for data and control-flow specification of such systems,

(ii) improving the overall model quality by early detection of requirement inconsistencies, possibly undesired non-deterministic behaviour, and possibly 'deadlock-esque' automata states,

(iii) stating a 'push-button' deployment concept taking non-functional requirements into account,

(iv) extending automatic deployment in terms of fault tolerance, and

(v) evaluating all concepts by dint of three different case studies.

## 9.2. Outlook

During the development phase of the COLA automotive approach and the accompanying IDE implementation several issues arose that were either not realised due to time constraints or due to outstanding conceptional work. In the following, these research questions are discussed.

**Requirements Engineering.** COLA supports adding requirement objects (textual informal descriptions or SALT formulae) to all COLA entities. This is useful for instance when generating the requirements specification documents. Nevertheless, having a linking mechanism to a central requirements management component would be preferable. Such a specific functionality to centrally store, maintain, and refine requirements would be appreciated. Then, traceability links between refined and original requirements could be established. Moreover, during development, engineers are oftentimes faced with the problem of contradicting,

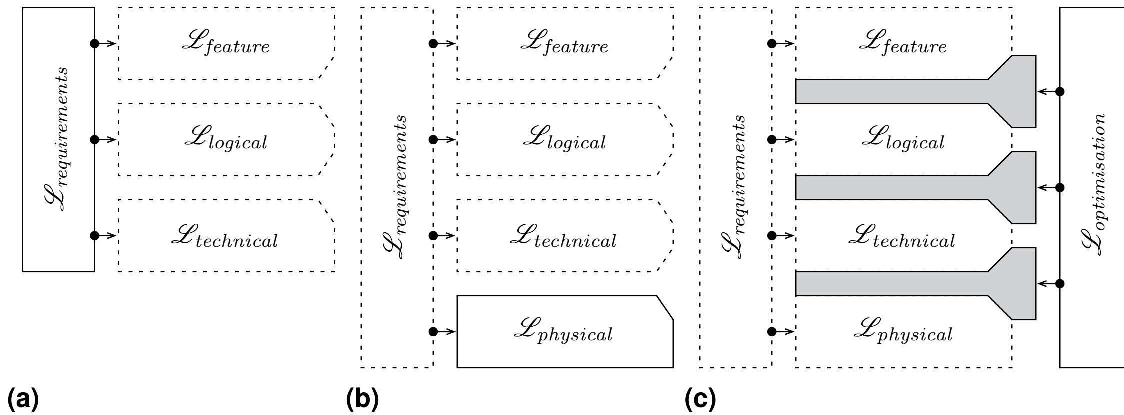



**Figure 9.1.:** Figures (a) to (c) extend the presented levels of abstraction to (a) model and organise requirements, (b) support of spatial modelling and analysis, and (c) comprehensive support of any kind of optimisation goals and models.

mutually exclusive requirements. At such point, engineers have to make a trade-off and finally decide which one to prefer. This weighting has to be documented, such that following projects can gain from this already done consideration of interests. All the mentioned points could be realised in a new conceptual layer—the Requirements Architecture $\mathscr{L}_{requirements}$ (cf. Figure 9.1a)—orthogonal to the already existing levels of abstraction. Besides the existing analysis support on the Feature Architecture, the Requirements Architecture would strive more towards the idea of 'front loading', i. e., a stronger emphasis on the early development phases with particular attention to quality assurance. To emphasis this phase even further, the addition of *parameterised* requirements attributes, such as for instance a desired braking distance, weight, or the bias-current would allow to feed back simulation results and facilitate early requirements verification.

**front loading**

Some non-functional requirements have been considered within this thesis, however for others like *safety*, *maintainability*, *extensibility*, *reusability* and other '*ilities*' a notion and way of modelling has to be developed first.

Concerning the requirements specification document generation process outlined in this thesis, one possible extension would be to provide an easy to use editor to change the proposed template document structure. Currently, an XSLT file has to be changed by hand.

**Computer Aided Design (CAD) and Spatial Modelling.** This dissertation proposed a way to solve the allocation problem, i. e., to determine on which ECU an executable software entity (cluster) should be executed under consideration of

optimisation constraints. What was not done, is to go one step further and ask where in the overall construction space an ECU should be placed. This again has to be performed with respect to optimisation and physical constraints. Conceivable considerations are to determine the placement with respect to

(i) harness routing (also harness routing alternatives could be considered) [189],

(ii) electromagnetic compatibility (harness, antennas, batteries, high-voltage network, power electronics, etc.) [73],

(iii) humidity and other influencing environmental factors such as temperature (consideration of try and wet area),

(iv) safety and redundancy aspects,

(v) vibrations, etc.

All the mentioned aspects could be modelled in a new architectural level—the Physical Architecture $\mathscr{L}_{physical}$, depicted in Figure 9.1b. The Physical Architecture could integrate features of current CAD (**C**omputer-**A**ided **D**esign) tools like PTC's Creo Elements/Pro, Dassault's CATIA, or Siemens PLM Software's NX or support interoperability.

Moreover, the COLA automotive approach, which is currently limited to software-intensive systems, could be extended towards a comprehensive modelling support for mechatronic systems (**mecha**nical engineering-elec**tronic** engineering), i. e., systems consisting of informatics, electronics, and mechanical parts. To do so, COLA needs to overcome the current limitation to only discrete modelling and has to provide modelling support of continuous systems.

**mechatronic system**

**Model-based Optimisation [127].** In the presented work, parametrisable optimisation goals were used during deployment. Besides other optimisation goals that are associated to the Physical Architecture, completely different optimisation goals and side condition could be taken into account. In the automotive context, related work for

(i) *cost models* for E/E architectures [16, 163] and those for modelling explicit uncertainties and risk analysis [17], as well as

(ii) *vendor selection* [15]

has been done. A new orthogonal layer—the Optimisation Architecture $\mathscr{L}_{optimisation}$—could subsume the mentioned and also other models for optimisation. The optimisation Architecture primarily influences the existing levels of abstraction during the transitions of a higher to the next lower level of abstraction. This circumstance is shown in Figure 9.1c. As a positive by-product optimisation goals are documented within the model.

**AUTOSAR.** As pointed out in Section 2.4.1, more and more OEMs proceed to demand AUTOSAR compatibility. The COLA-IDE, however, currently lacks AUTOSAR support. This stain can be removed with reasonable effort as discussed by Haberl [84].

# BIBLIOGRAPHY


[1] AUTOSAR: Automotive Open System Architecture. http://www.autosar.org. [Online; accessed 13-02-2012].

[2] GNU Linear Programming Kit, Version 4.41. http://www.gnu.org/software/glpk/glpk.html. [Online; accessed 12-05-2012].

[3] SCADE Suite. http://www.esterel-technologies.com/products/scade-suite/. [Online; accessed 12-05-2012].

[4] ISO 11898-1:2003 - Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, 2003.

[5] ISO 11898-2:2003 - Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit, 2003.

[6] ISO 11898-4:2004 - Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication, 2004.

[7] ISO 11898-3:2006 - Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium dependent interface, 2006.

[8] ISO 11898-5:2007 - Road vehicles – Controller area network (CAN) – Part 5: High-speed medium access unit with low-power mode, 2007.

[9] ISO/NP 11898-6 Road vehicles – Controller area network (CAN) – Part 6: High-speed medium access unit with selective wake-up functionality, under development.

[10] ACEA - European Automobile Manufacturers Association. Streamlining regulations: cost-effectiveness, impact assessments and harmonisation—the key to 'better regulation'. http://www.acea.be/index.php/news/

news_detail/streamlining_regulation/, 2009. [Online; accessed 06-07-2009].

[11] ADAC. *ADAC Motorwelt*, 5, 2008.

[12] ADAC. *ADAC Motorwelt*, 1:24, 2011.

[13] L. Amorim, P. R. M. Maciel, M. N. N. Jr., R. S. Barreto, and E. Tavares. Mapping live sequence chart to coloured petri nets for analysis and verification of embedded systems. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–25, 2006.

[14] S. Arthur, H. N. Breed, and C. Schmitt-Luehmann. Shifting car makeup shakes up OEM status quo: Software strength is critical. IBM White Paper. http://www.ibm.com/services/in/igs/pdf/g510-1692-00-shifting-car-makeup-shakes-up-oem-status-quo.pdf, 2003. [Online; accessed 07-09-2011].

[15] N. Arunkumar and L. Karunamoorthy. An optimization technique for vendor selection with quantity discounts using genetic algorithm. *Journal of Industrial Engineering International Islamic Azad University, South Teheran Branch*, Jan. 2007. [Online; accessed 07-09-2011].

[16] J. Axelsson. Cost models for electronic architecture trade studies. In *ICECCS*, pages 229–239. IEEE Computer Society, 2000.

[17] J. Axelsson. Cost models with explicit uncertainties for electronic architecture trade-off and risk analysis. In *Proc. 16th International Symposium of the International Council on Systems Engineering*, July 2006.

[18] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein. AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software. In *Proceedings of the SAE 2005 World Congress*, Detroit, MI, Apr. 2005. Society of Automotive Engineers.

[19] A. Bauer, M. Leucker, and J. Streit. Salt - structured assertion language for temporal logic. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.

[20] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In R. Lauwereins and J. Madsen, editors, *DATE*, pages 924–929. ACM, 2007.

[21] A. Benveniste, P. Caspi, P. Le Guernic, and N. Halbwachs. Data-flow synchronous languages. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 1–45. Springer Berlin / Heidelberg, 1994.

[22] Benz, Carl Friedrich. *Benz, Carl Friedrich: Lebensfahrt eines deutschen Erfinders. Die Erfindung des Automobils, Erinnerungen eines Achtzigjährigen*. Köhler und Amelang, 1936. Zenodot Verlagsgesellschaft mbH http://www.zeno.org.

[23] M. Bernard, C. Buckl, V. Döricht, M. Fehling, L. Fiege, H. von Grolman, N. Ivandic, C. Janello, C. Klein, K.-J. Kuhn, C. Patzlaff, B. C. Riedl, B. Schätz, and C. Stanek. Mehr Software (im) Wagen: Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilität der Zukunft. Technical report, fortiss GmbH, 2011.

[24] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[25] F. Bitsch and M. Gunzert. Formale verifikation von softwarespezifikationen in ASCET-SD und MATLAB. Technical report, IAS, Universität Stuttgart, 2000.

[26] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[27] P. Braun and M. Rappl. A model-based approach for automotive software development. In P. P. Hofmann and A. Schürr, editors, *OMER*, volume 5 of *LNI*, pages 100–105. GI, 2001.

[28] M. Broy. Automotive software and systems engineering (panel). In *MEMOCODE*, pages 143–149, 2005.

[29] M. Broy. Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM.

[30] M. Broy. Architecture based specification and verification of embedded software systems (work in progress). In Margaria and Steffen [147], pages 1–13.

[31] M. Broy. From system requirements documents to integrated system modeling artifacts. In U. M. Borghoff and B. Chidlovskii, editors, *ACM Symposium on Document Engineering*, page 98. ACM, 2009.

[32] M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild. Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München, 2008.

[33] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526 – 545, Apr. 2010.

[34] M. Broy, F. Huber, and B. Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 13(3):121–134, 1999.

[35] M. Broy, I. H. Krüger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[36] M. Broy, G. Reichart, and L. Rothhardt. Architekturen softwarebasierter Funktionen im Fahrzeug: von den Anforderungen zur Umsetzung. *Informatik Spektrum*, 34(1):42–59, 2011.

[37] M. Broy and K. Stølen. *Specification and development of interactive systems: Focus on streams, interfaces, and refinement*. Springer-Verlag, New York, 2001.

[38] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA, 2006. ACM.

[39] A. Burkert. Trends der Automobilelektronik und die Folgen für Autohersteller. http://www.atzonline.de/Aktuell/Nachrichten/1/15031/Trends-der-Automobilelektronik-und-die-Folgen-fuer-Autohersteller.html, Dec. 2011. [Online; accessed 28-03-2012].

[40] A. Campetelli, F. Hölzl, and P. Neubeck. User-friendly model checking integration in model-based development. In *The Twenty-Fourth International Conference on Computer Applications in Industry and Engineering (CAINE 2011)*,

Honolulu, Hawaii, USA, Nov. 2011. The International Society for Computers and Their Applications.

[41] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. N. och Dag. An industrial survey of requirements interdependencies in software product release planning. In *RE*, pages 84–93. IEEE Computer Society, 2001.

[42] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. In *LCTES*, pages 153–162. ACM, 2003.

[43] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[44] D. M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, Oct. 1984.

[45] D.-J. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg, F. Törner, and M. Törngren. Modelling support for design of safety-critical automotive embedded systems. In *SAFECOMP*, pages 72–85, 2008.

[46] S. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems–a brief survey. Technical report, University of Massachusetts, Amherst, MA, USA, 1987.

[47] M. B. Chrissis, M. Konrad, and S. Shrum. *CMMI(R): Guidelines for Process Integration and Product Improvement (2nd Edition) (The SEI Series in Software Engineering)*. Addison-Wesley Professional, 2006.

[48] M. Conrad and H. Dörr. Deployment of model-based software development in safety-related applications: Challenges and solutions scenarios. In H. C. Mayr and R. Breu, editors, *Modellierung*, volume 82 of *LNI*, pages 245–254. GI, 2006.

[49] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, Apr. 1968.

[50] CPN Tools. http://cpntools.org/. [Online; accessed 12-05-2012].

[51] Å. Dahlstedt and A. Persson. Requirements interdependencies: State of the art and future challenges. In *Engineering and Managing Software Requirements*, pages 95–116. Springer-Verlag, 2005.

[52] A. M. Davis. *Software requirements: analysis and specification*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1990.

[53] D. de Niz and P. H. Feiler. On resource allocation in architectural models. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 291–297. IEEE Computer Society, 2008.

[54] R. de Simone and C. André. Time modeling in marte. In *FDL*, pages 268–273. ECSI, 2007.

[55] V. Denner/Bosch/av. Gemeinsam mehr erreichen. http://www.all-electronics.de/track.php?p=1&ci=8127&ct=&l=http://www.all-electronics.de/media/file/8702, 2010. [Online; accessed 26-03-2012].

[56] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[57] E. W. Dijkstra. On the reliability of programs. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF. [Online; accessed 12-05-2012].

[58] E. W. Dijkstra. Chapter i: Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[59] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[60] M. Dinkel and U. Baumgarten. Modeling nonfunctional requirements: a basis for dynamic systems management. *SIGSOFT Softw. Eng. Notes*, 30(4):1–8, 2005.

[61] F. A. M. do Nascimento, M. F. S. Oliveira, and F. R. Wagner. Modes: Embedded systems design methodology and tools based on mde. In *Model-Based Methodologies for Pervasive and Embedded Software, 2007. MOMPES '07. Fourth International Workshop on*, pages 67–76, Mar. 2007.

[62] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[63] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, Aug. 2006.

[64] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[65] EAST-EEA. Electronics Architecture and Software Technology - Architecture Description Language, 2004. ITEA project 00009.

[66] M. Eberbach-Sahillioglu. Strukturanalyse und Wertschöpfungs- kette der deutschen Automobilindustrie. http://www.bic-kl.de/user/pdf/MA/automobilindustrie.pdf, 2004. [Online; accessed 28-March-2012].

[67] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42:42–52, 2009.

[68] H. Espinoza, H. Dubois, S. Gérard, J. L. M. Pasaje, D. C. Petriu, and C. M. Woodside. Annotating uml models with non-functional properties for quantitative analysis. In *MoDELS Satellite Events*, pages 79–90, 2005.

[69] ETAS Group. ASCET. http://www.etas.com/en/products/ascet_software_products.php. [Online; accessed 12-05-2012].

[70] European Software Institute. European User Survey Analysis, Report USV EUR 2.1, ESPITI Project, Jan. 1996.

[71] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[72] P. H. Feiler, B. Lewis, and S. Vestal. The SAE avionics architecture description language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES)*, Washington, DC, May 2003.

[73] R. Fellini, N. Michelena, P. Papalambros, and M. Sasena. Optimal design of automotive hybrid powertrain systems (invited). *Environmentally Conscious Design and Inverse Manufacturing, International Symposium on*, 0:400–405, Feb 1999.

[74] J. M. Fernandes, S. Tjell, J. B. Jorgensen, and O. Ribeiro. Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In *SCESM '07: Proceedings of the Sixth International Workshop on Scenarios and State Machines*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[75] FlexRay Consortium. FlexRay - the communication system for advanced automotive control applications. http://www.flexray.com/. [Online; accessed 07-09-2011].

[76] G. Fohler. Realizing changes of operational modes with pre run-time scheduled hard real-time systems. In *Proceedings of the Second International Workshop on Responsive Computer Systems*, Saitama, Japan, 1992.

[77] H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. In C. Choppy and O. Sokolsky, editors, *Monterey Workshop*, volume 6028 of *Lecture Notes in Computer Science*, pages 116–140. Springer, 2008.

[78] I. Galvao and A. Goknil. Survey of traceability approaches in model-driven engineering. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 313, Washington, DC, USA, 2007. IEEE Computer Society.

[79] M. R. Garey and D. S. Johnson. *Computers and Intractability (A Guide to Theory of NP-Completeness)*. Freeman, San Francisco, 1979.

[80] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[81] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proceedings of a conference on Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.

[82] F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, and H. M. W. Verbeek. Protos2cpn: using colored petri nets forr configuring and testing business processes. *STTT*, 10(1):95–110, 2008.

[83] K. Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society.

[84] W. Haberl. *Code Generation and System Integration of Distributed Automotive Applications*. Dissertation, Technische Universität München, München, 2011.

[85] W. Haberl, J. Birke, and U. Baumgarten. A Middleware for Model-Based Embedded Systems. In *Proceedings of the 2008 International Conference on Embedded Systems and Applications, ESA 2008*, Las Vegas, Nevada, USA, July 2008.

[86] W. Haberl, M. Herrmannsdoerfer, S. Kugele, M. Tautschnig, and M. Wechs. One click from model to reality, 2009. accepted for presentation at SAASE '09: Symposium on Automotive/Avionics Systems Engineering.

[87] W. Haberl, M. Herrmannsdoerfer, S. Kugele, M. Tautschnig, and M. Wechs. Seamless model-driven development put into practice. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 18–32. Springer, Oct. 2010.

[88] W. Haberl, S. Kugele, and U. Baumgarten. Reliable Operating Modes for Distributed Embedded Systems. In *Proceedings of the ICSE Workshop on Model-based Methodologies for Pervasive and Embedded Software*, volume 0, pages 11–21, Los Alamitos, CA, USA, May 2009. IEEE Computer Society.

[89] W. Haberl, S. Kugele, and U. Baumgarten. Model-Based Generation of Fault-Tolerant Embedded Systems. In H. R. Arabnia and A. M. G. Solo, editors, *Proceedings of the 2010 International Conference on Embedded Systems and Applications, ESA 2010*, pages 136–142, Las Vegas, Nevada, USA, July 2010. CSREA Press.

[90] W. Haberl, M. Tautschnig, and U. Baumgarten. From COLA Models to Distributed Embedded Systems Code. *IAENG International Journal of Computer Science*, 35(3):427–437, Sept. 2008.

[91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[92] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[93] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[94] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT*, pages 166–184, 2001.

[95] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 21–30, New York, NY, USA, 2005. ACM.

[96] M. Herrmannsdoerfer, W. Haberl, and U. Baumgarten. Model-level simulation for cola. In *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, pages 38–43, May 2009.

[97] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Business Process Management*, pages 220–235, 2005.

[98] C. A. R. Hoare. The emperor's old clothes. *Commun. ACM*, 24:75–83, Feb. 1981.

[99] A. Holzer, V. Januzaj, S. Kugele, B. Langer, C. Schallhart, M. Tautschnig, and H. Veith. Seamless testing for models and code. In D. Giannakopoulou and F. Orejas, editors, *FASE*, volume 6603 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2011.

[100] F. Hölzl and M. Feilkas. Autofocus 3: a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems*, MBEERTS'07, pages 317–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[101] IEEE. *IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)*. Institute of Electrical and Electronics Engineers, Inc., Oct. 1998. [Online; accessed 04-April-2012].

[102] International Electrotechnical Commission. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems (IEC 61508), 1998.

[103] International Organization for Standardization. Software Process Improvement and Capability Determination (ISO/IEC 15504-5, SPICE), 2006.

[104] International Organization for Standardization. Road vehicles–Functional safety (ISO 26262), 2011.

[105] D. Jackson. A direct path to dependable software. *Commun. ACM*, 52:78–88, Apr. 2009.

[106] V. Januzaj and S. Kugele. Model Analysis via a Translation Schema to Coloured Petri Nets. In D. Moldt, editor, *PNSE'09: Proceedings of the International Workshop on Petri Nets and Software Engineering*, pages 273–292, June 2009.

[107] K. Jensen. *Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use, volume 1*. Springer-Verlag, London, UK, 1996.

[108] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, volume 2*. Springer-Verlag, London, UK, 1997.

[109] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, volume 3*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[110] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[111] H. Kang, X. Yang, and S. Yuan. Modeling and verification of web services composition based on cpn. In *NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, pages 613–617, Washington, DC, USA, 2007. IEEE Computer Society.

[112] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[113] A. Kebemou and I. Schieferdecker. AutomotiveArchitect: An environment for the design of automotive systems architectures. In *6th International Conference on Informatics and Systems (INFOS 2008), Cairo, Egypt*, pages 40–47. Univ. Cairo, Faculty of Computers and Information, Mar. 2008.

[114] A. Kebemou and I. Schieferdecker. A model-based design approach for the partitioning of automotive systems. Fraunhofer Publica `http://publica.fraunhofer.de/oai.har`, 2008.

[115] M. U. Khan, K. Geihs, F. Gutbrodt, P. Gohner, and R. Trauter. Model-driven development of real-time systems with uml 2.0 and c. *Model-Based Methodologies for Pervasive and Embedded Software, International Workshop on*, 0:33–42, 2006.

[116] H. Kopetz. The time-triggered approach to real-time system design. In *Predictably Dependable Computing Systems, ESPRIT basic research series*, pages 53–66. Springer, 1995.

[117] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[118] H. Kopetz. The time-triggered architecture. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:22, 1998.

[119] H. Kopetz and G. Bauer. The time-triggered architecture. In *Proceedings of the IEEE*, volume 91, pages 112–126, 2003.

[120] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 09(1):25–40, 1989.

[121] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität Wien, Institut für Technische Informatik, Austria, 2004.

[122] F. Kordon, J. Hugues, and X. Renault. From model driven engineering to verification driven engineering. In U. Brinkschulte, T. Givargis, and S. Russo, editors, *SEUS*, volume 5287 of *Lecture Notes in Computer Science*, pages 381–393. Springer, 2008.

[123] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy*, pages 447–462. IEEE Computer Society, 2010.

[124] Kraftfahrt-Bundesamt. Jahresbericht 2009. http://www.KBA.de/cln_007/nn_124384/DE/Presse/Jahresberichte/jahresbericht__2009__pdf,templateId=raw,property=publicationFile.pdf/jahresbericht_2009_pdf.pdf, 2009. [Online; accessed 12-05-2012].

[125] L. M. Kristensen and K. Jensen. Specification and validation of an edge router discovery protocol for mobile ad hoc networks. In *SoftSpez Final Report*, pages 248–269, 2004.

[126] L. Kristenssen, J. Billington, and Z. Qureshi. Modelling military airborne mission systems for functional analysis. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 4A2/1–4A2/12 vol.1, Oct. 2001.

[127] S. Kugele. From model-based design to model-based optimization of embedded systems (extended abstract). In J. Strejček, editor, *Young Researchers Forum. Satellite workshop of MFCS & CSL*, pages 43–44, Brno, Czech Republic, Aug. 2010. DTU Informatics.

[128] S. Kugele and W. Haberl. Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*, Las Vegas, Nevada, USA, July 2008.

[129] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs. Optimizing automatic deployment using non-functional requirement annotations. In Margaria and Steffen [147], pages 400–414.

[130] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs. COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München, Sept. 2007.

[131] C. Kühnel, A. Bauer, and M. Tautschnig. Compatibility and reuse in component-based systems via type and unit inference. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 101–108, Washington, DC, USA, 2007. IEEE Computer Society.

[132] O. Labbani, J.-L. Dekeyser, and P. Boulet. Mode-automata based methodology for scade. In *In Hybrid Systems: Computation and Control (HSCC05*, pages 386–401, 2005.

[133] G. J. L. Lawrence, B. J. Hardy, J. A. Carroll, W. M. S. Donaldson, C. Visvikis, and D. A. Peel. A study on the feasibility of measures relating to the protection of pedestrians and other vulnerable road users - final report. `http://ec.europa.eu/enterprise/sectors/automotive/files/pagesbackground/pedestrianprotection/pedestrian_protection_study_en.pdf`, June 2004. [Online; accessed 12-05-2012].

[134] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sept. 1991.

[135] E. A. Lee. Computing needs time. *Commun. ACM*, 52:70–79, May 2009.

[136] E. A. Lee. Computing needs time. Technical Report UCB/EECS-2009-30, EECS Department, University of California, Berkeley, Feb. 2009.

[137] H. Legler, B. Gehrke, O. Krawczyk, U. Schasse, C. Rammer, N. Leheyda, and W. Sofka. Die Bedeutung der Automobilindustrie für die deutsche Volkswirtschaft im europäischen Kontext. ftp://ftp.zew.de/pub/zew-docs/gutachten/AutomobEndBericht_final.pdf, Sept. 2009. [Online; accessed 26-03-2012].

[138] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, 11(3):115–118, 1980.

[139] LIN Steering Group. LIN - Local Interconnect Network. http://www.lin-subbus.org/. [Online; accessed 07-09-2011].

[140] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[141] R. R. Lutz. Targeting safety-related errors during software requirements analysis. In *SIGSOFT FSE*, pages 99–106, 1993.

[142] D. A. Mackall and U. States. *Development and flight test experiences with a flight-crucial digital control system [microform] / Dale A. Mackall*. National Aeronautics and Space Administration, Scientific and Technical Information Division, Washington, DC, 1988.

[143] D. MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, Cambridge, MA, USA, 2001.

[144] W. H. Maisel, M. O. Sweeney, W. G. Stevenson, K. E. Ellison, and L. M. Epstein. Recalls and safety alerts involving pacemakers and implantable cardioverter-defibrillator generators. *JAMA*, 286(7):793–9, Aug. 2001.

[145] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In C. Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 185–199. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0053571.

[146] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[147] T. Margaria and B. Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, volume 17 of *Communications in Computer and Information Science*. Springer, 2008.

[148] S. Matic, M. Goraczko, J. Liu, D. Lymberopoulos, B. Priyantha, and F. Zhao. Resource modeling and scheduling for extensible embedded platforms. Technical Report MSR-TR-2006-176, Microsoft Reasearch, One Microsoft Way, Redmond, WA, USA, 2006.

[149] Mercer Management Consulting and HypoVereinsbank. Studie-Automobiletechnologie 2010, Aug. 2001.

[150] A. Metzner and C. Herde. Rtsat–an optimal and efficient approach to the task allocation problem in distributed architectures. In *RTSS*, pages 147–158, 2006.

[151] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems. www.misra.org.uk, Oct. 2004.

[152] J. Mössinger. Software in automotive systems. *IEEE Software*, 27(2):92–94, 2010.

[153] MOST Cooperation. MOST – Media Oriented Systems Transport. http://www.mostcooperation.com/. [Online; accessed 07-09-2011].

[154] N. Navet and F. Simonot-Lion. *The Automotive Embedded Systems Handbook*. Industrial Information Technology Series. Taylor & Francis / CRC Press, 2008.

[155] Object Management Group. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). OMG document: ptc/07-08-04, 2007.

[156] Object Management Group. OMG Systems Modeling Language (OMG SysML™). OMG document: formal/2010-06-01, 2010.

[157] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[158] L. C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[159] L. Petrucci, J. Billington, L. M. Kristensen, and Z. H. Qureshi. Developing a formal specification for the mission system of a maritime surveillance aircraft. In *ACSD '03: Proceedings of the Third International Conference on Application of Concurrency to System Design*, pages 92–101, Washington, DC, USA, June 2003. IEEE Computer Society.

[160] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[161] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, Apr. 2006.

[162] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering (FOSE '07)*, pages 55–71, 2007.

[163] C. Quigley, R. McMurran, R. Jones, and P. Faithfull. An investigation into cost modelling for design of distributed automotive electrical architectures. In *Automotive Electronics, 2007 3rd Institution of Engineering and Technology Conference on*, pages 1–9, June 2007.

[164] Z. H. Qureshi. Formal modelling and analysis of mission-critical software in military avionics systems. In *SCS '06: Proceedings of the eleventh Australian workshop on Safety critical systems and software*, pages 67–77, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[165] R. Racu, A. Hamann, R. Ernst, and K. Richter. Automotive software integration. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 545–550. IEEE, 2007.

[166] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards. Requirements traceability: Theory and practice. *Annals of Software Engineering*, 3:397–415, 1997. 10.1023/A:1018969401055.

[167] G. Reichart and H. Heinecke. Systemintegration - im Wechselspiel von Architektur, Technologie und Prozess. In *10. Jahrestagung Euroforum*, 2006.

[168] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[169] J. W. Reuter. Analysis and comparison of 3 code generation tools. In *Proceedings of the SAE 2004 World Congress*, Detroit, MI, Mar. 2004. Society of Automotive Engineers.

[170] S. Rittmann. *A methodology for modeling usage behavior of multi-functional systems*. PhD thesis, Institut für Informatik, Technische Universität München, 2009.

[171] J. Romberg. *Synthesis of distributed systems from synchronous dataflow programs*. PhD thesis, Technische Universität München, 2006.

[172] J. Romberg and A. Bauer. Loose Synchronization of Event-Triggered Networks for Distribution of Synchronous Programs. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*, pages 193–202, Pisa, Italy, Sept. 2004. Association for Computing Machinery.

[173] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *Proceedings of the 19th USENIX Security Symposium*, Aug. 2010.

[174] RTCA DO-178B. Software considerations in airborne systems and equipment certification, 1992.

[175] J. M. Rushby. Bus architectures for safety-critical embedded systems. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 306–323, London, UK, 2001. Springer-Verlag.

[176] J. M. Rushby. Automated formal methods enter the mainstream. *J. UCS*, 13(5):650–660, 2007.

[177] A. Sangiovanni-Vincentelli and M. D. Natale. Embedded system design for automotive applications. *Computer*, 40:42–51, 2007.

[178] B. Schätz. Model-based engineering of embedded control software. In *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 53–62, Washington, DC, USA, 2006. IEEE Computer Society.

[179] J. Schäuffele and T. Zurawka. *Automotive Software Engineering*. Vieweg + Teubner, Wiesbaden, 4. edition, 2010.

[180] S. Schulz, J. W. Rozenblit, and K. Buchenrieder. Multilevel testing for design verification of embedded systems. *IEEE Design & Test of Computers*, 19(2):60–69, 2002.

[181] SEI AADL Team. OSATE. An extensible Source AADL Tool Environment. Technical report, Software Engineering Institute, Carnegie Mellon University, 2004.

[182] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[183] S. Sentilles, A. Pettersson, D. Nyström, T. Nolte, P. Pettersson, and I. Crnkovic. Save-IDE—A tool for design, analysis and implementation of component-based embedded systems. In *ICSE*, pages 607–610. IEEE, 2009.

[184] Standard ML. http://www.standardml.org/. [Online; accessed 12-05-2012].

[185] Standish Group. Software Chaos, 1995.

[186] I. Stürmer, D. Weinberg, and M. Conrad. Overview of existing safeguarding techniques for automatically generated code. *SIGSOFT Softw. Eng. Notes*, 30:1–6, May 2005.

[187] The EP and the Council of the EU. Directive 2005/66/EC: relating to the use of frontal protection systems on motor vehicles. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2005:309:0037:0054:EN:PDF, 2005. [Online; accessed 06-07-2009].

[188] The Mathwork Inc. *Using Simulink*, 2012. [Online; accessed 12-05-2012].

[189] D. D. Turner. Determining the optimal distributed electronic module solution of an automotive system while incorporating harness routing alternatives in an electrical/electronic architecture tool environment. In *SAE World Congress & Exhibition*. SAE International, Apr. 2008.

[190] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[191] M. von der Beeck. Development of logical and technical architectures for automotive systems. *Software and System Modeling*, 6(2):205–219, 2007.

[192] Z. Wang, W. Haberl, S. Kugele, and M. Tautschnig. Automatic Generation of SystemC Models from Component-based Designs for Early Design Validation and Performance Analysis. In *Proceedings of the 7th International Workshop on Software and Performance, WOSP 2008*, pages 23–26, Princeton, NJ, USA, June 2008. ACM.

[193] Z. Wang, A. Herkersdorf, S. Merenda, and M. Tautschnig. A model driven development approach for implementing reactive systems in hardware. In *FDL*, pages 197–202. IEEE, 2008.

[194] Z. Wang, A. Sanchez, and A. Herkersdorf. Scisim: a software performance estimation framework using source code instrumentation. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 33–42, New York, NY, USA, 2008. ACM.

[195] J. Weber. *Automotive Development Processes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[196] M. Weber and J. Weisbrod. Requirements engineering in automotive development - experiences and challenges. In *RE*, pages 331–340. IEEE Computer Society, 2002.

[197] D. Wild, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, and S. Rittmann. An architecture-centric approach towards the construction of dependable automotive software. In *Proc. of the SAE 2006 World Congress, Detroit*. Society of Automotive Engineers, Apr. 2006.

[198] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[199] R. Wuyts and S. Ducasse. Non-functional requirements in a component model for embedded systems. In *SAVCBS 2001*, 2001.

[200] D. Wybo and D. Putti. A qualitative analysis of automatic code generation tools for automotive powertrain applications. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 225–230, 1999.

[201] Y. Yang, Q. Tan, Y. Xiao, F. Liu, and J. Yu. Transform BPEL workflow into hierarchical CP-nets to make tool support for verification. In *APWeb*, pages 275–284, 2006.

[202] P. Zave. Classification of research efforts in requirements engineering. *ACM Comput. Surv.*, 29(4):315–321, 1997.

[203] P. Zave. *An experiment in feature engineering*, pages 353–377. Springer-Verlag, New York, NY, USA, 2003.

[204] P. Zave and R. T. Yeh. Executable requirements for embedded systems. In *ICSE*, pages 295–304, 1981.

[205] W. Zheng, Q. Zhu, M. D. Natale, and A. S. Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 161–170, Washington, DC, USA, 2007. IEEE Computer Society.

[206] D. Zowghi and V. Gervasi. The Three Cs of Requirements: Consistency, Completeness, and Correctness. In *Proceedings of 8th International Workshop on Requirements Engineering: Foundation for Software Quality, (REFSQ'02)*, 2002.

# LIST OF FIGURES


1.1. Multi-dimensional explosion of complexity . . . . . . . . . . . . . . 2
1.3. Proportion of electronics-incurred costs . . . . . . . . . . . . . . . . 4
1.4. E/E value of software and hardware . . . . . . . . . . . . . . . . . 5
1.5. Structure of this thesis . . . . . . . . . . . . . . . . . . . . . . . . 9

2.1. Structure of a control system . . . . . . . . . . . . . . . . . . . . . 15
2.2. Discrete core of COLA systems . . . . . . . . . . . . . . . . . . . . 15
2.3. Worst-case execution time (WCET) . . . . . . . . . . . . . . . . . . 16
2.4. The Benz Patent-Motorwagen Nr. 3 of 1888 . . . . . . . . . . . . . . 18
2.5. Milestones in automotive E/E history . . . . . . . . . . . . . . . . . 19
2.6. Car recalls in Germany between 1998 and 2009 . . . . . . . . . . . . 22
2.7. Reasons and evolution of car breakdowns in Germany . . . . . . . 23
2.8. Characteristic of the automotive domain . . . . . . . . . . . . . . . . 24
2.9. Worldwide automobile production per year . . . . . . . . . . . . . . 25
2.10. Typical automotive domains . . . . . . . . . . . . . . . . . . . . . . 28
2.11. Example of a seat module . . . . . . . . . . . . . . . . . . . . . . . 31
2.12. Different AUTOSAR layers . . . . . . . . . . . . . . . . . . . . . . . 34
2.13. Rapid increase of the number of ECUs . . . . . . . . . . . . . . . . 36
2.14. Feature interaction example . . . . . . . . . . . . . . . . . . . . . . 37

3.1. Hierarchy of abstraction levels . . . . . . . . . . . . . . . . . . . . . 44
3.2. Traceability links between artefacts . . . . . . . . . . . . . . . . . . 45
3.3. Tool integration gap . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.4. Architecture of the COLA engineering environment . . . . . . . . . 52

4.1. Examples showing the FODA and COLA's feature notation . . . . 58
4.2. Hierarchical decomposition of COLA models . . . . . . . . . . . . . 62
4.3. Conceptual class diagram of COLA's Hardware Architecture . . . 62

4.4. Exemplary clustering of a Logical Architecture . . . . . . . . . . . . 63
4.5. COLA's Technical Architecture . . . . . . . . . . . . . . . . . . . . . 65
4.6. Different levels of abstraction in the COLA automotive approach . 66
4.7. Graphical notation of the basic arithmetic operators . . . . . . . . . 69
4.8. Graphical notation of the basic comparison operators . . . . . . . . 69
4.9. IPO-model of high-level COLA design . . . . . . . . . . . . . . . . 70
4.10. COLA automaton modelling a data-flow if . . . . . . . . . . . . . 71
4.11. Operating modes modelled using different formalisms . . . . . . . 72
4.12. Exemplary hierarchical decomposition using operating modes . . . 73
4.13. Exemplary textual COLA syntax . . . . . . . . . . . . . . . . . . . 74
4.14. Exemplary Bessel filter diagram . . . . . . . . . . . . . . . . . . . . 76
4.15. Exemplary input and output signals of the Bessel filter . . . . . . . 77
4.16. Floating body in a magnetic field . . . . . . . . . . . . . . . . . . . 78
4.17. PID controller with a controlled system . . . . . . . . . . . . . . . 78
4.18. COLA deployment steps . . . . . . . . . . . . . . . . . . . . . . . . 80

5.1. The verification-driven engineering helicoidal life cycle . . . . . . . 86
5.2. SALT to Büchi automata transformation steps . . . . . . . . . . . . . 90
5.3. Conjunction of the presented SALT specifications . . . . . . . . . . 91
5.4. Different fields of application of COLA automata . . . . . . . . . . 93
5.5. Simultaneously enabled automaton transitions . . . . . . . . . . . . 94
5.6. Deterministic embedded automaton . . . . . . . . . . . . . . . . . . 95
5.7. Backwards-search to restrict over-approximation . . . . . . . . . . . 96
5.8. Pruning the search space by deriving new predicates . . . . . . . . 96
5.9. Verification result visualised within the COLA-IDE . . . . . . . . . 100
5.10. CPN for a COLA delay block initialised with 1 . . . . . . . . . . . . 105
5.11. Graphical models of a functional block in COLA and CPN notation 105
5.12. Exemplary COLA network . . . . . . . . . . . . . . . . . . . . . . . 107
5.13. Exemplary hierarchical CPN model . . . . . . . . . . . . . . . . . . 108
5.14. High-level automaton in COLA and CPN notation . . . . . . . . . 110
5.15. Screenshot of the COLA simulator . . . . . . . . . . . . . . . . . . 112
5.16. CPN model of the running example . . . . . . . . . . . . . . . . . . 114

6.1. Containment of a scope of services . . . . . . . . . . . . . . . . . . . 122
6.2. Differentiation between customer and system requirements . . . . 123
6.3. Generic document structure . . . . . . . . . . . . . . . . . . . . . . . 124
6.4. Context of the scope of services of 'LR_SmartOpener' . . . . . . . . 125
6.5. Specification document generation and visualisation process . . . . 126
6.6. Exemplary structure and layout of the generated document . . . . 128

6.7. Semantically tangible requirements specification document . . . . . 134

7.1. Systems compiler . . . . . . . . . . . . . . . . . . . . . . . . . . 136
7.2. Scheduling cycle . . . . . . . . . . . . . . . . . . . . . . . . . . 137
7.3. System Architecture . . . . . . . . . . . . . . . . . . . . . . . . 138
7.4. Separation of architectural and resource model . . . . . . . . . . . . 140
7.5. Classification of non-functional requirements . . . . . . . . . . . . . 142
7.6. Terminology for periodic scheduling . . . . . . . . . . . . . . . . . 148
7.7. Scheduling taxonomy . . . . . . . . . . . . . . . . . . . . . . . . 148
7.8. Possibilities for data-flow in COLA networks . . . . . . . . . . . . . 151
7.9. Cluster dependency graph of the ACC case study . . . . . . . . . . 152
7.10. Middleware API . . . . . . . . . . . . . . . . . . . . . . . . . . 154
7.11. Reduced cluster dependency graph . . . . . . . . . . . . . . . . . 156
7.12. Exemplary run of Algorithm 7 . . . . . . . . . . . . . . . . . . . . 159
7.13. Multi-rate scheduling example . . . . . . . . . . . . . . . . . . . 163
7.14. Determination of initial bounds . . . . . . . . . . . . . . . . . . . 164
7.15. Convergence of $b$ for every operating mode . . . . . . . . . . . . . 168
7.16. Visualisation of the number of search iterations . . . . . . . . . . . 169
7.17. Fault tolerance through re-allocation of clusters . . . . . . . . . . . 171
7.18. Exemplary re-allocation of clusters . . . . . . . . . . . . . . . . . 174
7.19. COLA-IDE deployment perspective . . . . . . . . . . . . . . . . . 180

8.1. Focused activities of the case studies . . . . . . . . . . . . . . . . 182
8.2. Simulation of the ACC behaviour . . . . . . . . . . . . . . . . . . 185
8.3. Hardware topology of the LEGO Mindstorms ACC case study . . . 186
8.4. LEGO Mindstorms ACC . . . . . . . . . . . . . . . . . . . . . . . 186
8.5. Operating modes of the multi-ECU APS case study . . . . . . . . . 187
8.6. APS functionality in detail . . . . . . . . . . . . . . . . . . . . . 188
8.7. COLA simulation view of the APS case study . . . . . . . . . . . . 189
8.8. Hardware Topology of the APS case study . . . . . . . . . . . . . . 189
8.9. Image of the APS case study with schedule plan . . . . . . . . . . . 192
8.10. Feature Architecture of the CHO example . . . . . . . . . . . . . . 193
8.11. Chain of effects for the feature ‘no key’ . . . . . . . . . . . . . . . 195

9.1. Extension of the levels of abstraction . . . . . . . . . . . . . . . . 200

A.1. PID controller COLA model . . . . . . . . . . . . . . . . . . . . . 239
A.2. PID controller with three different parameter sets . . . . . . . . . . 240

# LIST OF TABLES

2.1. Characteristics of different automotive domains . . . . . . . . . . . 29

3.1. Commonly used COTS tools . . . . . . . . . . . . . . . . . . . . . . . . 51

5.1. Identifier used within the exemplary SALT formulae . . . . . . . . . 88

7.1. Attributes of clusters and ECUs used for the allocation . . . . . . . 141
7.2. Comparison between static and dynamic hard real-time scheduling 150
7.3. Multi-rate periodic cluster scheduling . . . . . . . . . . . . . . . . . 163

8.1. Components of the APS's Hardware Topology . . . . . . . . . . . . 190

# LIST OF ALGORITHMS

1. Checks whether a given automaton $\mathcal{A}$ is deterministic . . . . . . . . . 98
2. Translation schema for functional blocks . . . . . . . . . . . . . . . . . 106
3. Translation schema for networks . . . . . . . . . . . . . . . . . . . . . 109

4. Translation schema for automata . . . . . . . . . . . . . . . . . . . . . 116
5. COLA2CPN translation algorithm . . . . . . . . . . . . . . . . . . . . . 117
6. Generates a requirements specification document . . . . . . . . . . . 126
-. Procedure descend(Unit $u$) performs a DFS . . . . . . . . . . . . . . . . 127

7. Computation of the set of schedule sets $\mathbf{S}$ . . . . . . . . . . . . . . . . . 158
8. Calculates a solution for the optimal scheduling problem $\mathscr{S}^*$ . . . . 167
9. Redundant allocation of safety-critical clusters . . . . . . . . . . . . . 173

# GLOSSARY

## A

**ACC** Short for Adaptive Cruise Control (ACC). The ACC system is an advanced driver assistance system, which automatically controls the speed of a vehicle by holding a safe distance to a vehicle driving ahead at a specified velocity., p. 14.

**APS** Short for Autonomous Parking Assistant (APS). In German 'Parkmanöver Assistent (PMA)'. It is a advanced driver assistance systems for automatic parking., p. 185.

**ASIL** Short for Automotive Safety Integrity Level (ASIL). ISO 26262 provides an automotive-specific risk-based approach for determining risk classes (ASIL levels). Depending on the severity (S), exposure (E), and controllability (C), the respective ASIL level can be looked-up in a table., p. 33.

**Automotive SPICE** Automotive SPICE is an automotive domain-specific variant of the international standard ISO/IEC 15504 (SPICE). It is aimed at benchmarking the development processes of ECU suppliers., p. 45.

**AUTOSAR** AUTOSAR is a development partnership between all major automobile, electronic control unit manufacturers, and engineering tool providers. The partnership is aimed at facilitating software exchange between different ECUs. AUTOSAR provides a uniform software architecture with standardised description and configuration formats., p. 34.

**AUTOSIG** Short for Automotive Special Interest Group (AUTOSIG). In AUTOSIG, vendors like AUDI, BMW, Daimler, Porsche, Volkswagen, Fiat,

Ford, and others are represented., p. 45.

## B

**BrickOS** BrickOS is an alternative operating system for the LEGO Mindstorms RCX micro-computer. Applications for it can be written in the programming languages C and C++., p. 184.

## C

**CAN** Short for Controller Area Network (CAN). CAN is a standardised vehicle bus originally developed by Robert Bosch GmbH in 1983., p. 30.

## D

**DO-178B** Software Considerations in Airborne Systems and Equipment Certification is a guidance for software development published by RTCA, Incorporated. The standard was developed by RTCA and EUROCAE. The FAA accepts use of DO-178B as a means of certifying software in avionics., p. 19.

**DSL** Short for Domain-Specific Language (DSL). A domain-specific language (DSL) is a programming language dedicated to a specific problem domain such as COLA is a DSL for the automotive domain., p. 81.

## E

**ECU** Short for Electronic Control Unit (ECU). An ECU is a generic term in the automotive domain. It is used to talk about embedded control systems., p. 2.

**EDF** Short for Earliest Deadline First (EDF). It assigns priorities dynamically according to the deadline: a task receives the highest priority if its deadline is the earliest amongst the set of all ready tasks., p. 149.

**ESC** Short for Electronic Stability Control (ESC or ESP). This system improves the vehicle's stability by detecting loss of steering control and selective wheel breaking., p. 32.

## F

**feature interaction** A feature interaction is some way in which a feature or features modify or influence another feature in defining the overall system behaviour [203]., p. 37.

**FlexRay** The FlexRay bus is a serial, deterministic, and fault-tolerant automotive bus-system developed by the FlexRay Consortium. Main design goals were to be faster and more reliable than CAN and TTP., p. 33.

## G

**GENIVI Alliance** The GENIVI Alliance is a non-profit consortium consisting of major OEMs, semiconductor and device manufacturer, as well as software companies. There goal is to provide a range of compliance statements, and a compliance programme for GENIVI certification for automotive infotainment systems as well as a Linux-based open source reference platform including operating system and middleware., p. 34.

## H

**HMI** Short for Human-Machine Interface (HMI). It is the space where interaction between humans and machines occurs., p. 14.

## I

**ISO 26262** The ISO 26262 standard 'Road vehicles - Functional safety' is an adaptation of the Functional Safety standard IEC 61508 for automotive E/E systems., p. 33.

## L

**LEGO Mindstorms** The LEGO Mindstorms series provides a programmable Brick computer as controller, where sensors and actuators can be attached to build programmable robots. Moreover, a visual programming environment is provided., p. 184.

**LIN** Short for Local Interconnect Network (LIN). The LIN bus is a low-cost serial automotive networking bus-system. Its aim is to connect intelli-

gent sensors and actuator devices to the remaining E/E architecture., p. 30.

## M

**Model-driven development (MDD)** Model-driven development (MDD) is a software development methodology, which focuses on the creation and exploration of models oftentimes in the context of specific domains., p. 41.

**MOST** Short for Media Oriented Systems Transport (MOST). MOST is a high-speed multimedia network usually used for infotainment systems., p. 30.

## N

**NFR** Short for Non-Functional Requirements (NFR). NFRs (or product qualities) are defined in the international standard for the evaluation of software quality ISO/IEC 9126 Software engineering., p. 139.

## O

**OEM** Short for Original Equipment Manufacturer (OEM)., p. 3.

## P

**PDM** Short for Product Data Model (PDM)., p. 48.

## R

**real-time system** A real-time system (RTS) is a computing system underlying specific real-time constraints. Its correctness does not only depend on the functional or logical correctness, but also on its timely execution., p. 16.

**RTOS** Short for Real-Time Operating System, p. 138.

## S

**SEooC** Short for Safety Element out of Context (SEooC). When developing generic components without a concrete application, no safety goals are defined, yet. Those components can be developed as so-called SEooC referring to ISO 26262 by making and documenting assumptions., p. 38.

## T

**TT CAN** Short for Time-Triggered CAN (TT CAN)., p. 33.

## W

**WCET** Short for Worst-Case Execution Time (WCET). The WCET is the maximum length a computational task could take when it is executed on a specific hardware platform., p. 16.

APPENDIX

# A

# PID Controller

This chapter gives details on the PID controller modelled in COLA (cf. Section 4.3.4). In Section A.1 the COLA model is given, and Section A.2 shows the simulation results.

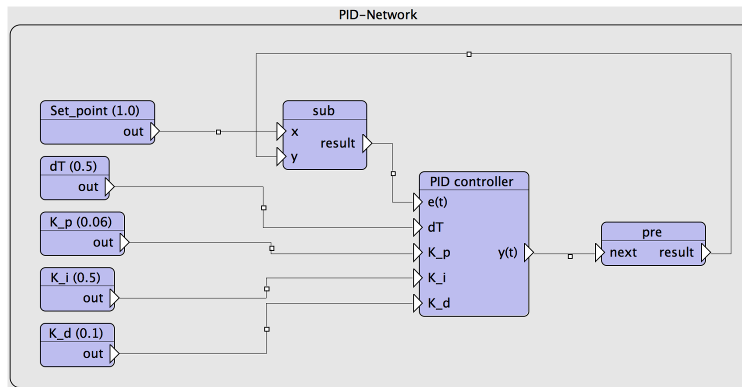
## A.1. COLA model

This section presents details on the PID controller's COLA model. Figure A.1a gives the high-level view of the PID controller model. Basically the controller's functionality is encapsulated in the COLA network 'PID controller'. The constants are placed on the left side of the 'PID-Network'. They are used to parameterise the controller using $dT$ (sample rate), $K_p$, $K_i$, and $K_d$—these are the proportional, integral, and derivative gains—, and finally the setpoint is specified. The delay block 'pre' feeds back the new value $y(t)$ to compute the error, i. e., the difference between the desired value und the current value. Figure A.1b gives details on the 'PID-controller' network. There, $y(t)$ is computed by adding the proportional, the integral, and the derivative ratios. Each of them is amplified by the specified input gains. The implementations for the 'integral' and 'derivative' networks are given in the Figures A.1c and A.1d.

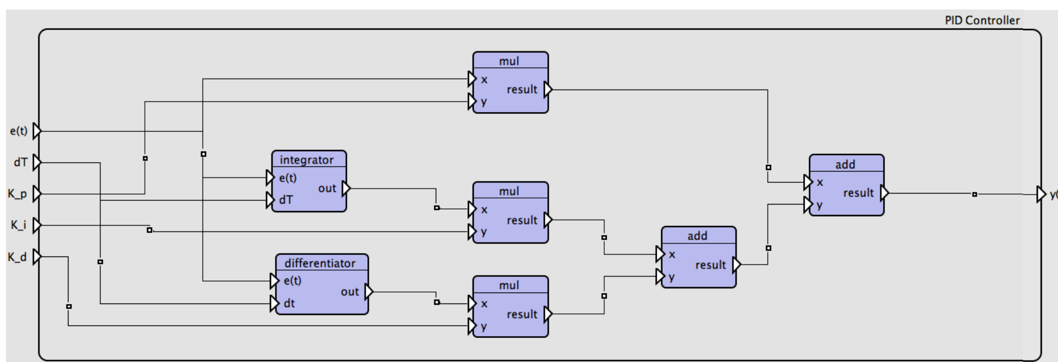
## A.2. Simulation with different parameter sets

In Figure A.2 the valuation of the modelled controller is depicted with three different input parameter sets. Each time, the setpoint and the sampling rate is set

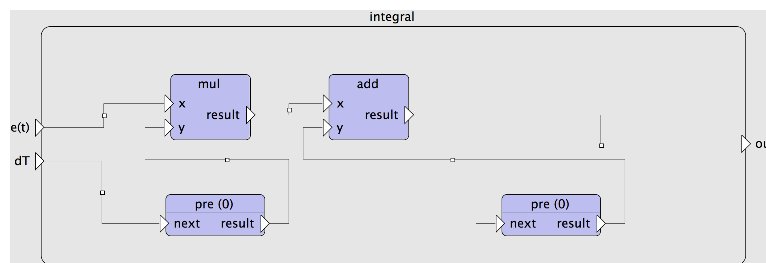
to $1.0$ and $0.5$, respectively. The gains $K_i$ (integral) and $K_d$ (derivative) are varied. The values are gained using the COLA-IDE's simulation capabilities.



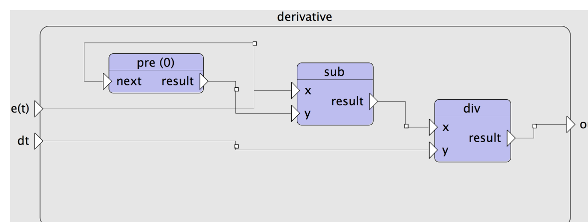

**(a)** PID network




**(b)** PID controller




**(c)** Integral




**(d)** Derivative

**Figure A.1.:** These figures give details of the PID controller modelled in COLA. Figure (a) depicts the high-level view of the model. (b) shows the PID controller as a glass box. Figures (c) and (d) show the implementation of the integral and the derivative components.

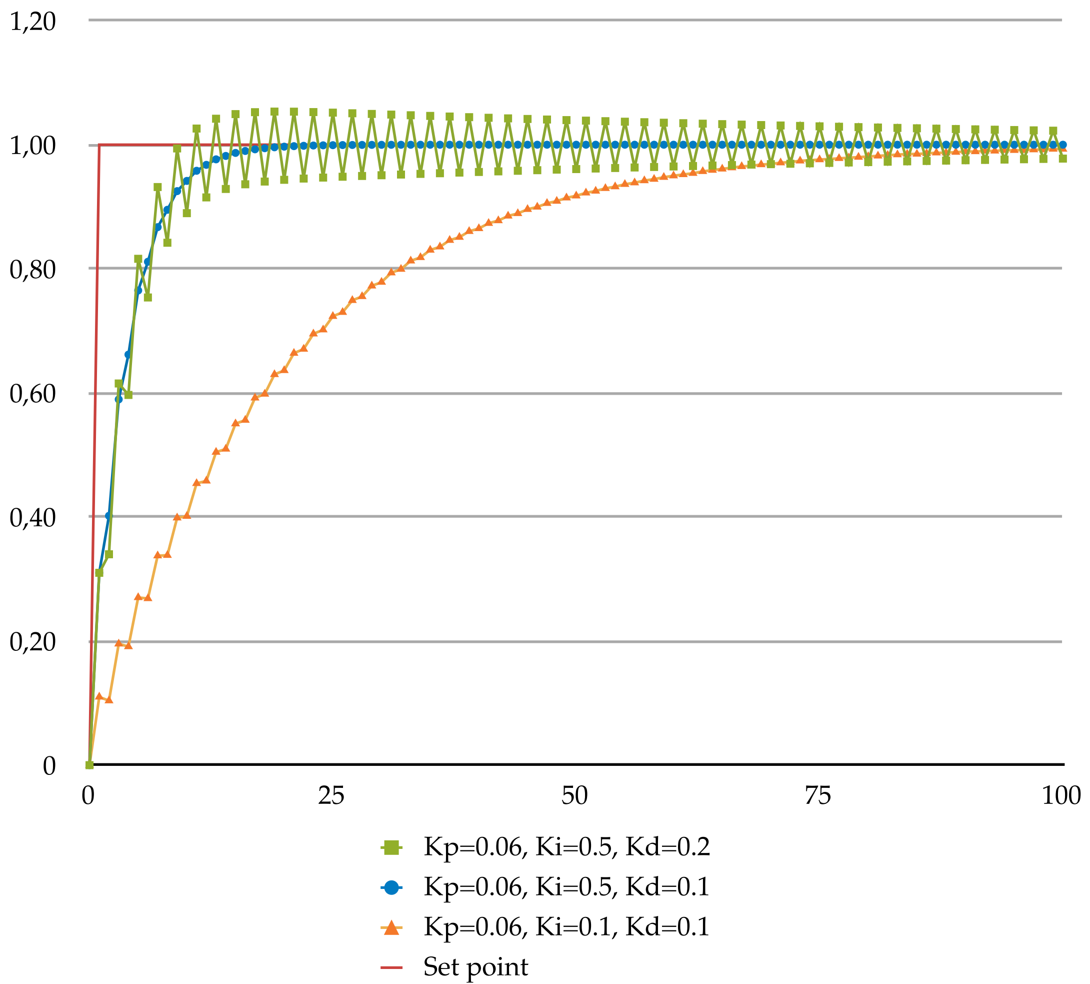



**Figure A.2.:** PID controller with three different parameter sets.