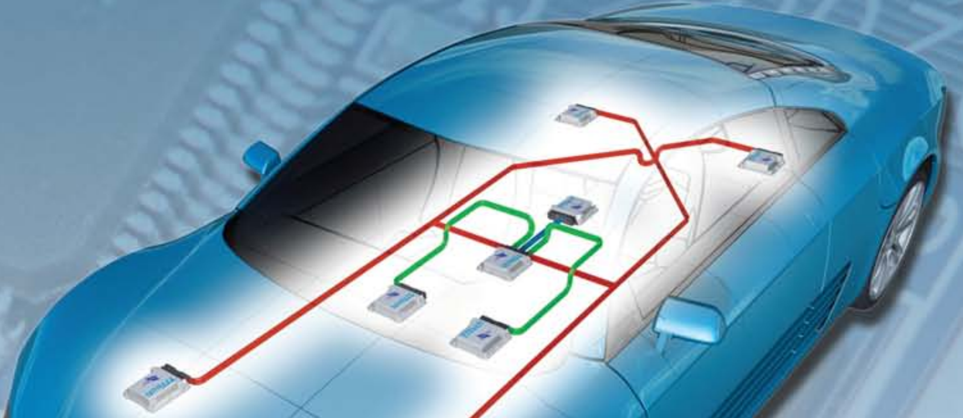
INDUSTRIAL INFORMATION TECHNOLOGY SERIES

Automotive Embedded Systems Handbook



Edited by

Nicolas Navet

Françoise Simonot-Lion

CRC Press
Taylor & Francis Group

Automotive Embedded Systems Handbook

INDUSTRIAL INFORMATION TECHNOLOGY SERIES

Series Editor
RICHARD ZURAWSKI

Automotive Embedded Systems Handbook
Edited by Nicolas Navet and Françoise Simonot-Lion

Integration Technologies for Industrial Automated Systems
Edited by Richard Zurawski

Electronic Design Automation for Integrated Circuits Handbook
Edited by Luciano Lavagno, Grant Martin, and Lou Scheffer

Embedded Systems Handbook
Edited by Richard Zurawski

Industrial Communication Technology Handbook
Edited by Richard Zurawski

INDUSTRIAL INFORMATION TECHNOLOGY SERIES

Automotive Embedded Systems Handbook

Edited by
Nicolas Navet
Françoise Simonot-Lion

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-0-8493-8026-6 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Automotive embedded systems handbook / edited by Nicolas Navet and
Francoise Simonot-Lion.
p. cm. -- (Industrial information technology ; 5)
Includes bibliographical references and index.
ISBN-13: 978-0-8493-8026-6
ISBN-10: 0-8493-8026-X
1. Automotive computers. 2. Automobiles--Electronic equipment. 3. Automobiles--Automatic control--Equipment and supplies. 4. Embedded computer systems. I. Navet, Nicolas. II. Simonot-Lion, Francoise. III. Title. IV. Series.

TL272.53.A9868 2009
629.2--dc22 2008024406

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Part I Automotive Architectures

Part II Embedded Communications

Part III Embedded Software and Development Processes

Part IV Verification, Testing, and Timing Analysis

Preface

The objective of the *Automotive Embedded Systems Handbook* is to provide a comprehensive overview about existing and future automotive electronic systems. The distinctive features of the automotive world in terms of requirements, technologies, and business models are highlighted and state-of-the-art methodological and technical solutions are presented in the following areas:

- In-vehicle architectures
- Multipartner development processes (subsystem integration, product line management, etc.)
- Software engineering methods
- Embedded communications
- Safety and dependability assessment: validation, verification, and testing

The book is aimed primarily at automotive engineering professionals, as it can serve as a reference for technical matters outside their field of expertise and at practicing or studying engineers, in general. On the other hand, it also targets research scientists, PhD students, and MSc students from the academia as it provides them with a comprehensive introduction to the field and to the main scientific challenges in this domain.

Over the last 10 years, there has been an exponential increase in the number of computer-based functions embedded in vehicles. Development processes, techniques, and tools have changed to accommodate that evolution. A whole range of electronic functions, such as navigation, adaptive control, traffic information, traction control, stabilization control, and active safety systems, are implemented in today's vehicles. Many of these new functions are not stand-alone in the sense that they need to exchange information—and sometimes with stringent time constraints—with other functions. For example, the vehicle speed estimated by the engine controller or by wheel rotation sensors needs to be known in order to adapt the steering effort, to control the suspension, or simply to choose the right wiper speed. The complexity of the embedded architecture is continually increasing. Today, up to 2500 signals (i.e., elementary information such as the speed of the vehicle) are exchanged through up to 70 electronic control units (ECUs) on five different types of networks.

One of the main challenges of the automotive industry is to come up with methods and tools to facilitate the integration of different electronic subsystems coming from various suppliers into the vehicle's global electronic architecture. In the last 10 years, several industry-wide projects have been undertaken in that direction (AEE*, EAST, AUTOSAR, OSEK/VDX, etc.) and significant results have already been achieved (e.g., standard components such as operating systems, networks and middleware, "good practices," etc.). The next step is to build an accepted open software architecture, as well as the associated development processes and tools, which should allow for easily integrating the different functions and ECUs provided by carmakers and third-part suppliers. This is ongoing work in the context of the AUTOSAR project.

As all the functions embedded in cars do not have the same performance or safety needs, different qualities of service are expected from the different subsystems. Typically, an in-car embedded system is divided into several functional domains that correspond to different features and constraints. Two of them are concerned specifically with real-time control and safety in the vehicle's behavior: the "power train" (i.e., control of engine and transmission) and the "chassis" (i.e., control of suspension, steering, and braking) domains. For these safety-critical domains, the technical solutions must ensure that the system is dependable (i.e., able to deliver a service that can be justifiably trusted) while being cost-effective at the same time.

These technical problems are very challenging, in particular due to the introduction of X-by-wire functions, which replace the mechanical or hydraulic systems, such as braking or steering, with electronic systems. Design paradigms (time-triggered, "safety by construction"), communication networks (FlexRay, TTP/C), and middleware layers (AUTOSAR COM) are currently being actively developed in order to address these needs for dependability.

The principal players in the automotive industry can be divided into:

- Vehicle manufacturers
- Automotive third-part suppliers
- Tool and embedded software suppliers

The relationships between them are very complex. For instance, suppliers providing key technologies are sometimes in a very strong position and may impose their technical approach on carmakers. Since the competition is fierce among carmakers and suppliers, keeping the company's know-how confidential is crucial. This has strong implications in the technical field. For instance, the validation of the system (i.e., verifying that the system meets its constraints) may have to be carried out

* Architecture Electronique Embarquée (AEE, 1997–2000) is a French project supported by the Ministry of Industry with PSA and Renault, Sagem, Siemens, and Valeo as main industrial partners. Its main objective was to find solutions for easing the portability of applicative level software. Embedded Electronic Architecture (EAST-EEA, 2001–2004, see http://www.east-eea.net/) is an European ITEA project involving most major European carmakers, automotive third-part suppliers, tools and middleware suppliers, and research institutes. Automotive Open Architecture (AUTOSAR, 2004–2007, see http://www.autosar.org) is an ongoing follow-up to EAST-EAA aimed at establishing open standards for automotive embedded architecture. Open systems and the corresponding interfaces for automotive electronics (OSEK, see http://www.osek-vdx.org) is a German automotive industry project defining standards for software components used for communication, network management, and operating systems. Some of the outcomes of OSEK (e.g., OSEK/OS) are already widely used in production cars.

with techniques that do not require full knowledge of the design rationales and implementation details.

Shortening the time to market puts on added pressure because carmakers must be able to propose their innovations—that usually rely heavily on electronic systems—within a time frame that allows for these innovations to be really considered as innovative. The players involved strive to reduce the development time while the system's overall complexity increases, demanding even more time. This explains why, despite the economic competition, they have agreed to work together to define standard components and reference architecture that will help cut overall development time.

This book contains 15 contributions, written by leading experts from industry and academia directly involved in the engineering and research activities treated in this book. Many of the contributions are from industry or industrial research establishments at the forefront of the automotive domain: Siemens (Germany), ETAS (Germany), Volvo (Sweden), Elektrobit (Finland), Carmeq (Germany), The MathWorks Inc. (United States), and Audi (Germany). The contributions from academia and research organizations are presented by renowned institutions such as Technical University of Berlin (Germany), LORIA-Nancy University (France), INRIA (France), IRCCyN Nantes University (France), KTH (Sweden), Mälardalen University (Sweden), Kettering University (United States), University of Aveiro (Portugal), and Ulm University (Germany).

Organization

Automotive Architectures

This part provides a broad introduction to automotive embedded systems, their design constraints, and AUTOSAR as the emerging de facto standard. Chapter 1, "Vehicle Functional Domains and Their Requirements," introduces the main functions embedded in a car and how these functions are divided into functional domains (chassis, power train, body, multimedia, safety, and human–machine interfaces). Some introductory words describe the specificities of the development process as well as the requirements in terms of safety, comfort, performance, and cost that need to be taken into account.

In Chapter 2, "Application of the AUTOSAR Standard," the authors tackle the problem of the standardization of in-vehicle embedded electronic architectures. They analyze the current status of software in the automotive industry and present the specifications elaborated within the AUTOSAR consortium in terms of standardization. Particular attention has to be paid to AUTOSAR because it is becoming a standard that everyone has to understand and deal with.

Finally, Chapter 3, "Intelligent Vehicle Technologies," presents the key technologies that have been developed to meet today's, and tomorrow's, automotive challenges in terms of safety, better use of energy, and better use of space, especially in cities. These technologies, such as sophisticated sensors (radar, stereo-vision, etc.), wireless networks, or intelligent driving assistance, will facilitate the conception of partially or fully autonomous vehicles that will reshape the transport landscape and commuters' travel experience in the twenty-first century.

Embedded Communications

The increasing complexity of electronic architectures embedded in a vehicle, and locality constraints for sensors and actuators, has led the automotive industry to adopt a distributed approach for implementing the set of functions. In this context, networks and protocols are of primary importance. They are the key support for integrating functions, reducing the cost and complexity of wiring, and furnishing a means for fault tolerance. Their impact in terms of performance and dependability is crucial as a large amount of data is made available to the embedded functions through the networks. This part includes three chapters dedicated to networks and protocols.

Chapter 4, "A Review of Embedded Automotive Protocols," outlines the main protocols used in automotive systems; it presents the features and functioning schemes of CAN, J1850, FlexRay, TTCAN, and the basic concepts of sensor/actuator networks (LIN, TTP/A) and multimedia networks (MOST, IDB1394). The identification of the communication-related services commonly offered by a middleware layer and an overview of the AUTOSAR proposals conclude the chapter.

CAN is at present the network that is the most widely implemented in vehicles. Nevertheless, despite its efficiency and performance, CAN does not possess all the features that are required for safety-critical applications. The purpose of the chapter, "Dependable Automotive CANs," is to point out CAN's limitations, which reduce dependability, and to present technical solutions to overcome or minimize these limitations. In particular, the authors describe techniques, protocols, and architectures based on CAN that improve the dependability of the original protocol in some aspects while still maintaining a high level of flexibility, namely (Re)CANcentrate, CANELy, FTT-CAN, and FlexCAN.

With the development of technology, there has been an increasing number of functions with strong needs in terms of data bandwidth. In addition, safety requirements have become more and more stringent. To answer to both of these constraints, in 2000, the automotive industry began to develop a new protocol—FlexRay. Chapter 5 "FlexRay Protocol," explains the rationale of FlexRay and gives a comprehensive overview of its features and functioning scheme. Finally, an evaluation of the impact of FlexRay on the development process concludes the chapter.

Embedded Software and Development Processes

The design process of an electronic-embedded system relies on a tight cooperation between car manufacturers and suppliers under a specific concurrent engineering approach. Typically, carmakers provide the specification of the subsystems to suppliers, who are then in charge of the design and realization of these subsystems, including the software and hardware components, and possibly the mechanical or hydraulic parts. The results are furnished to the carmakers, who in turn integrate them into the car and test them. Then comes the "calibration" phase, which consists of tuning control and regulation parameters in order to meet the required performances of the controlled systems. Any error detected during the integration phase leads to costly corrections in the specification or design steps. For this reason, in order to improve the effectiveness of the development process, new design methodologies are emerging, in particular, the concept of a virtual platform, which is now gaining acceptance in the area of the electronic automotive systems design.

The virtual platform concept requires modeling techniques that are suited to the design and validation activities at each step of the development process. In this context, model-based development (MBD) has been extensively studied by both car manufacturers and suppliers. How to adapt this approach to the automotive industry is discussed in Chapter 10, "Model-Based Development of Automotive Embedded Systems." This chapter identifies the benefits of model-based development, explores the state of practice, and looks into the major challenges for the automotive industry.

One of the main issues in automotive systems is to reduce the time to market. The reuse of components, or of subsystems, is one way to achieve this objective. In Chapter 8, "Reuse of Software in Automotive Electronics," the authors give an overview of the challenges faced when reusing software in the automotive industry, the different viewpoints on the reuse issue of manufacturers and suppliers, and the impact of the multipartner development approach.

Sharing the same modeling language between the different parties involved in development is an effective means to ease the cooperative development process. The main purpose of such a language is, on the one hand, to support the description of the system at the different steps of its development (requirement specification, functional specification, design, implementation, tuning, etc.) according to the different points of view and, on the other hand, to ensure a consistency between these different views. Another important aspect is its ability to reflect the structure of the embedded systems as an architecture of components (hardware components, functional components, software components). The ideas and principles brought by architecture description languages (ADLs) are well suited to these objectives. What is an ADL? Why are ADLs needed? What are the main existing ADLs and their associated tools? What are the main ongoing projects in the automotive context? Answers to these questions can be found in Chapter 9 "Automotive Architecture Description Languages."

The introduction and management of product lines is of primary importance for the automotive industry. These product lines are linked to mechanical system variations, and certain customer-visible variations, offered in a new car. The purpose of Chapter 7, "Product Lines in Automotive Electronics" is to present the systematic planning and continuous management of variability throughout the development process. This chapter provides some techniques on how to model the variability as well as traceability guidelines for the different phases of development.

Verification, Testing, and Timing Analysis

Some functions in a car are critical from the safety point of view, such as, for example, certain functions in the chassis or the power train domain. Thus, validation and verification are of primary importance.

Testing is probably the most commonly used verification technique in the automotive industry. A general view on testing approaches is given in Chapter 11 "Testing Automotive Control Software." In particular, this chapter describes current practices and several methods that are involved in the testing activities, such as the classification-tree method, test scenario selection approaches, and black-box/white-box testing processes. As already mentioned, communication networks and protocols are key factors for the dependability and performance of an embedded system. Hence, certain properties on communication architectures have to be verified. Chapter 12, "Testing and Monitoring of FlexRay-Based Applications," deals with the application of testing techniques to the FlexRay protocol. The authors review the constraints in the validation step in the development process of automotive applications and explain how fault-injection and monitoring techniques can be used for testing FlexRay.

As CAN is the most popular network embedded in cars, its evaluation has been the subject of a long line of research. Chapter 13, "Timing Analysis of CAN-Based Automotive Communication Systems," summarizes the main results that have been obtained over the last 15 years in the field of timing analysis on CAN. In particular, it is explained how to calculate bounds on the delays that frames experience before arriving at the receiver end (i.e., the response times of the frames). Accounting for the occurrence of transmission errors, for instance due to electromagnetic interferences, is also covered in this chapter. Due to its medium access control protocol based on the priorities of the frames, CAN possesses good real-time characteristics. However, a shortcoming that becomes increasingly problematic is its limited bandwidth. One solution that is being investigated by car manufacturers is to schedule the messages with offsets, which leads to a desynchronization of the message streams. As shown in Chapter 14, "Scheduling Messages with Offsets on Controller Area Network: A Major Performance Boost," this "traffic shaping" strategy is very beneficial in terms of worst-case response times. The experimental results suggest that sound offset strategies may extend the life span of CAN further, and may defer the introduction of FlexRay and additional CANs.

Chapter 15 "Formal Methods in the Automotive Domain: The Case of TTA," describes the formal verification research done in the context of time-triggered architecture (TTA), and more specifically the work that concerns time-triggered protocol (TTP/C), which is the core underlying communication network of the TTA. These formal verification efforts have focused on crucial algorithms in distributed systems: clock synchronization, group membership algorithm, or the startup algorithm, and have led to strong results in terms of dependability guarantees. To the best of our knowledge, TTA is no longer being considered or implemented in cars. Nevertheless, the experience gained over the years with the formal validation of the TTA will certainly prove to be extremely valuable for other automotive communication protocols such as FlexRay, especially in the perspective that certification procedures will be enforced for automotive systems, as they are now for avionic systems.

We would like to express our gratitude to all of the authors for the time and energy they have devoted to presenting their topic. We are also very grateful to Dr. Richard Zurawski, editor of the Industrial Information Technology Series, for his continuous support and encouragements. Finally, we would like to thank CRC Press for having agreed to publish this book and for their assistance during the editorial process.

We hope that you, the readers of this book, will find it an interesting source of inspiration for your own research or applications, and that it will serve as a reliable, complete, and well-documented source of information for automotive-embedded systems.

Nicolas Navet

Françoise Simonot-Lion

Editors

Nicolas Navet has been a researcher at the Grand Est Research Centre at the National Institute for Research in Computer Science and Control (INRIA), Nancy, France, since 2000. His research interests include real-time scheduling, the design of communication protocols for real-time and fault-tolerant data transmission, and dependability evaluation when transient faults may occur (e.g., EMI). He has authored more than 70 refereed publications and has received the CAN in Automation International Users and Manufacturers Group research award in 1997 as well as five other distinctions (e.g., best paper awards). Since 1996, he has worked on numerous contracts and projects with automotive manufacturers and suppliers. He is the founder and chief scientific officer of RealTime-at-Work, a company dedicated to providing services and software tools that help optimize the hardware resource utilization and verify that dependability constraints are met. He holds a BS in computer science from the University of Berlin, Berlin, Germany and a PhD in computer science from the Institut National Polytechnique de Lorraine, Nancy, France.

Françoise Simonot-Lion is a professor of computer science at University of Nancy, Nancy, France. She has been the scientific leader of the Real Time and InterOperability (TRIO) research team since 1997, which is an INRIA project at the Lorraine Laboratory of Computer Science Research and Applications (LORIA) in Nancy, France. From 2001 to 2004, she was responsible for CARAMELS, a joint research team with PSA Peugeot Citroën funded by the French Ministry for Research and Technology. She has participated in the French Embedded Electronic Architecture project (AEE, 1999–2001), and in the European project ITEA EAST-EEA (2001–2004). The purpose of ITEA EAST was to define an industry-wide layered software architecture, including a communication middleware, and a common architecture description language supporting a formal description of in-vehicle embedded systems (EAST-ADL). She is also an associate editor of *IEEE Transactions on Industrial Informatics*.

Contributors

Luis Almeida
Department of Electronics
Telecommunication and Informatics
University of Aveiro
Aveiro, Portugal

Jakob Axelsson
Volvo Car Corporation
Gothenburg, Sweden

and

Department of Computer Engineering
Mälardalen University
Västeras, Sweden

Manuel Barranco
Department of Mathematics and Informatics
University of the Balearic Islands
Palma, Spain

Patrice Bodu
Informatics, Mathematics and Automation for La Route Automatisée
National Institute for Research in Computer Science and Control (INRIA)
Rocquencourt, France

DeJiu Chen
Department of Machine Design
Royal Institute of
Technology
Stockholm, Sweden

Mirko Conrad
The MathWorks, Inc.
Natick, Massachusetts

Joaquim Ferreira
Department of Information Technologies Engineering
Polytechnic Institute of Castelo Branco
Castelo Branco, Portugal

Ines Fey
Safety and Modeling Consultants
Berlin, Germany

Ulrich Freund
ETAS
Stuttgart, Germany

Thomas M. Galla
Elektrobit Corporation
Vienna, Austria

Michael Golm
Siemens AG
Princeton, New Jersey

Michael Gonschorek
Elektrobit Corporation
Munich, Germany

Mathieu Grenier
Lorraine Laboratory of Computer Science Research and Applications
Nancy, France

and

University of Nancy
Nancy, France

Hans A. Hansson
Mälardalen Real-Time Research Centre
Mälardalen University
Västeras, Sweden

Bernd Hardung
AUDI AG
Ingolstadt, Germany

Lionel Havet
National Institute for Research in Computer Science and Control (INRIA)
Nancy, France

and

RealTime-at-Work
Nancy, France

Thorsten Kölzow
AUDI AG
Ingolstadt, Germany

Andreas Krüger
AUDI AG
Ingolstadt, Germany

Christian Kühnel
Faculty of Informatics
Technical University of Munich
Garching, Germany

Henrik Lönn
Volvo Technology Corporation
Gothenburg, Sweden

Diana Malvius
Department of Machine Design Royal Institute of Technology
Stockholm, Sweden

Nicolas Navet
National Institute for Research in Computer Science and Control (INRIA)
Nancy, France

and

RealTime-at-Work
Nancy, France

Mikael Nolin
Mälardalen Real-Time Research Centre
Mälardalen University
Västeras, Sweden

Thomas Nolte
Mälardalen Real-Time Research Centre
Mälardalen University
Västeras, Sweden

Roman Pallierer
Elektrobit Corporation
Vienna, Austria

Michel Parent
Informatics, Mathematics and Automation for La Route Automatisée
National Institute for Research in Computer Science and Control (INRIA)
Rocquencourt, France

Holger Pfeifer
Institute of Artificial Intelligence
Ulm University
Ulm, Germany

Juan Pimentel
Electrical and Computer Engineering Department
Kettering University
Flint, Michigan

Julian Proenza
Department of Mathematics and Informatics
University of the Balearic Islands
Palma, Spain

Sasikumar Punnekkat
Mälardalen Real-Time Research Centre
Mälardalen University
Västeras, Sweden

Mark-Oliver Reiser
Software Engineering Group
Technical University of Berlin
Berlin, Germany

Guillermo Rodriguez-Navas
Department of Mathematics and Informatics
University of the Balearic Islands
Palma, Spain

Bernard Sanchez
Continental Automotive GmbH
Toulouse, France

Bernhard Schätz
Faculty of Informatics
Technical University of Munich
Garching, Germany

Françoise Simonot-Lion
Lorraine Laboratory of Computer Science Research and Applications
Nancy, France

and

University of Nancy
Nancy, France

Friedhelm Stappert
Continental Automotive GmbH
Regensburg, Germany

Martin Törngren
Department of Machine Design
Royal Institute of Technology
Stockholm, Sweden

Yvon Trinquet
Institute of Communications Research and Cybernetics of Nantes (IRCCyN)
Nantes, France

and

University of Nantes
Nantes, France

Stefan Voget
Continental Automotive GmbH
Regensburg, Germany

Matthias Weber
Carmeq GmbH
Berlin, Germany

I

Automotive Architectures

1

Vehicle Functional Domains and Their Requirements

Françoise Simonot-Lion
Lorraine Laboratory of Computer Science Research and Applications

Yvon Trinquet
Institute of Communications Research and Cybernetics of Nantes

1.1 General Context

The automotive industry is today the sixth largest economy in the world, producing around 70 million cars every year and making an important contribution to government revenues all around the world [1]. As for other industries, significant improvements in functionalities, performance, comfort, safety, etc. are provided by electronic and software technologies. Indeed, since 1990, the sector of embedded electronics, and more precisely embedded software, has been increasing at an annual rate of 10%. In 2006, the cost of an electronic-embedded system represented at least 25% of the total cost of a car and more than 35% for a high-end model [2]. This cost is equally shared between electronic and software components. These general trends have led to currently embedding up to 500 MB on more than 70 microprocessors [3] connected on communication networks. The following are some of the various examples. Figure 1.1 shows an electronic architecture embedded in a Laguna (source: Renault French carmaker) illustrating several computers interconnected and controlling the engine,

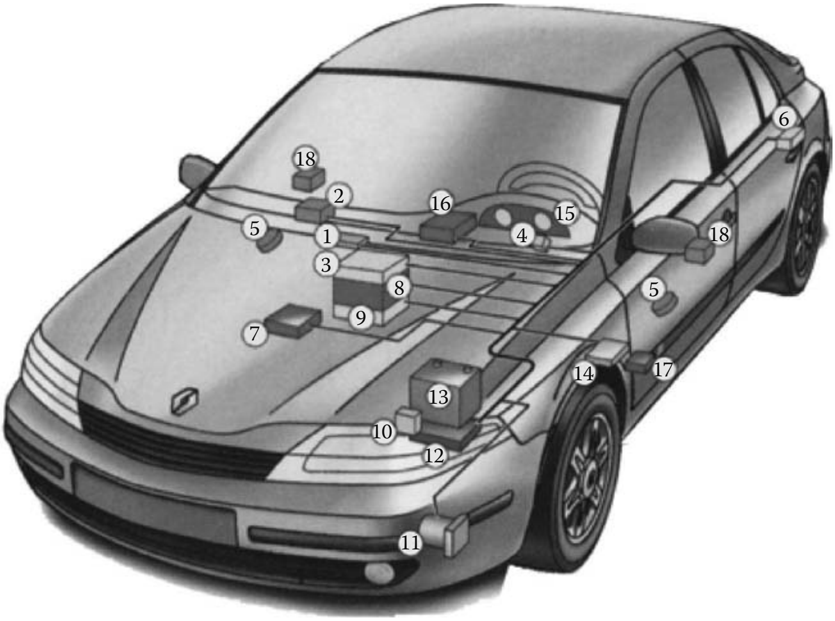


FIGURE 1.1 A part of the embedded electronic architecture of a Renault Laguna. (Courtesy of Renault Automobile. With permission.)

the wipers, the lights, the doors, and the suspension or providing a support for interaction with the driver or the passengers. In 2004, the embedded electronic system of a Volkswagen Phaeton was composed of more than 10,000 electrical devices, 61 microprocessors, three controller area networks (CAN) that support the exchanges of 2500 pieces of data, several subnetworks, and one multimedia bus [4]. In the Volvo S70, two networks support the communication between the microprocessors controlling the mirrors, those controlling the doors and those controlling the transmission system and, for example, the position of the mirrors is automatically controlled according to the sense the vehicle is going and the volume of the radio is adjusted to the vehicle speed, information provided, among others, by the antilock braking system (ABS) controller. In a recent Cadillac, when an accident causes an airbag to inflate, its microcontroller emits a signal to the embedded global positioning system (GPS) receiver that then communicates with the cell phone, making it possible to give the vehicle's position to the rescue service. The software code size of the Peugeot CX model (source: PSA Peugeot Citroen French carmarker) was 1.1 KB in 1980, and 2 MB for the 607 model in 2000. These are just a few examples, but there are many more that could illustrate this very large growth of embedded electronic systems in modern vehicles.

The automotive industry has evolved rapidly and will evolve even more rapidly under the influence of several factors such as pressure from state legislation, pressure from customers, and technological progress (hardware and software aspects). Indeed, a great surge for the development of electronic control systems came through the regulation concerning air pollution. But we must also consider the pressure from consumers for more performance (at lower fuel consumption), comfort, and safety. Add to all this the fact that satisfying these needs and obligations is only possible because of technological progress.

Electronic technology has made great strides and nowadays the quality of electronic components—performance, robustness, and reliability—enables using them even for critical systems. At the same time, the decreasing cost of electronic technology allows them to be used to support any function in a car. Furthermore, in the last decade, several automotive-embedded networks such as local interconnect networks (LIN), CAN, TTP/C, FlexRay, MOST, and IDB-1394 were developed. This has led to the concept of multiplexing, whose principal advantage is a significant reduction in the wiring cost as well as the flexibility it gives to designers; data (e.g., vehicle speed) sampled by one microcontroller becomes available to distant functions that need them with no additional sensors or links.

Another technological reason for the increase of automotive embedded systems is the fact that these new hardware and software technologies facilitate the introduction of functions whose development would be costly or not even feasible if using only mechanical or hydraulic technology. Consequently, they allow to satisfy the end user requirements in terms of safety, comfort, and even costs. Well-known examples are electronic engine control, ABS, electronic stability program (ESP), active suspension, etc. In short, thanks to these technologies, customers can buy a safe, efficient, and personalized vehicle, while carmakers are able to master the differentiation between product variations and innovation (analysts have stated that more than 80% of innovation, and therefore of added value, will be obtained thanks to electronic systems [5]). Furthermore, it also has to be noted that some functions can only be achieved through digital systems. The following are some examples: (1) the mastering of air pollution can only be achieved by controlling the engine with complex control laws; (2) new engine concepts could not be implemented without an electronic control; (3) modern stability control systems (e.g., ESP), which are based on close interaction between the engine, steering, and braking controllers, can be efficiently implemented using an embedded network.

Last, multimedia and telematic applications in cars are increasing rapidly due to consumer pressure; a vehicle currently includes electronic equipment like hand-free phones, audio/radio devices, and navigation systems. For the passengers, a lot of entertainment devices, such as video equipment and communication with the outside world are also available. These kinds of applications have little to do with the vehicle's operation itself; nevertheless they increase significantly as part of the software included in a car.

In short, it seems that electronic systems enable limitless progress. But are electronics free from any outside pressure? No. Unfortunately, the greatest pressure on electronics is cost!

Keeping in mind that the primary function of a car is to provide a safe and efficient means of transport, we can observe that this continuously evolving "electronic revolution" has two primary positive consequences. The first is for the customer/consumer, who requires an increase in performance, comfort, assistance for mobility efficiency (navigation), and safety on the one hand, while on the other hand, is seeking reduced fuel consumption and cost. The second positive consequence is for the stakeholders, carmakers, and suppliers, because software-based technology reduces marketing time, development cost, production, and maintenance cost. Additionally, these innovations have a strong impact on our society because reduced fuel consumption and exhaust emissions improve the protection of our natural resources and the environment, while the introduction of vision systems, driver assistance, onboard diagnosis, etc., targets a "zero death" rate, as has been stated in Australia, New Zealand, Sweden, and the United Kingdom.

However, all these advantages are faced with an engineering challenge; there have been an increasing number of breakdowns due to failure in electric/electronic systems. For example, Ref. [6] indicates that, for 2003, 49.2% of car breakdowns were due to such problems in Germany. The quality of a product obviously depends on the quality of its development, and the increasing complexity of in-vehicle embedded systems raises the problem of mastering their development. The design process is based on a strong cooperation between different players, in particular Tier 1 suppliers and carmakers, which involves a specific concurrent engineering approach. For example, in Europe or Japan, carmakers provide the specification for the subsystems to suppliers, who, in turn, compete to find a solution for these carmakers. The chosen suppliers are then in charge of the design and realization of these subsystems, including the software and hardware components, and possibly the mechanical or hydraulic parts as well. The results are furnished to the carmakers, or original equipment manufacturer (OEM), who install them into the car and test them. The last step consists of calibration activities where the control and regulation parameters are tuned to meet the required performance of the controlled systems. This activity is closely related to the testing activities. In the United States, this process is slightly different since the suppliers cannot really be considered as independent from the carmakers.

Not all electronic systems have to meet the same level of dependability as the previous examples. While with a multimedia system customers require a certain quality and performance, with a chassis control system, safety assessment is the predominant concern. So, the design method for each subsystem depends on different techniques. Nevertheless, they all have common distributed characteristics and they must all be at the level of quality fixed by the market, as well as meeting the safety requirements and the cost requirements. As there has been a significant increase in computer-based and distributed controllers for the core critical functions of a vehicle (power train, steering or braking systems, "X-by-wire" systems, etc.) for several years now, a standardization process is emerging for the safety assessment and certification of automotive-embedded systems, as has already been done for avionics and the nuclear industry, among others. Therefore, their development and their production need to be based on a suitable methodology, including their modeling, a priori evaluation and validation, and testing. Moreover, due to competition between carmakers or between suppliers to launch new products under cost, performance, reliability, and safety constraints, the design process has to cope with a complex optimization problem.

In-vehicle embedded systems are usually classified according to domains that correspond to different functionalities, constraints, and models [7–9]. They can be divided among "vehicle-centric" functional domains, such as power train control, chassis control, and active or passive safety systems and "passenger centric" functional domains where multimedia/telematics, body/comfort, and human–machine interface (HMI) can be identified.

1.2 Functional Domains

Carmakers distinguish several domains for embedded electronics in a car, even though sometimes the membership of only one domain for a given compartment is not easy to justify. According to the glossary of the European ITEA EAST-EEA project [10], a domain is defined as "a sphere of knowledge, influence, and activity in which one or more systems are to be dealt with (e.g., are to be built)." The term domain can be used as a means to group mechanical and electronic systems.

Historically, five domains were identified: power train, chassis, body, HMI, and telematics. The power train domain is related to the systems that participate in the longitudinal propulsion of the vehicle, including engine, transmission, and all subsidiary components. The chassis domain refers to the four wheels and their relative position and movement; in this domain, the systems are mainly steering and braking. According to the EAST-EEA definition, the body domain includes the entities that do not belong to the vehicle dynamics, thus being those that support the car's user, such as airbag, wiper, lighting, window lifter, air conditioning, seat equipment, etc. The HMI domain includes the equipment allowing information exchange between electronic systems and the driver (displays and switches). Finally, the telematic domain is related to components allowing information exchange between the vehicle and the outside world (radio, navigation system, Internet access, payment).

From one domain to another, the electronic systems often have very different features. For example, the power train and chassis domains both exhibit hard real-time constraints and a need for high computation power. However, the hardware architecture in the chassis domain is more widely distributed in the vehicle. The telematic domain presents requirements for high data throughput. From this standpoint, the technological solutions used are very different, for example, for the communication networks, but also for the design techniques and verification of the embedded software.

1.2.1 Power Train Domain

As mentioned previously, this domain represents the system that controls the engine according to requests from the driver (e.g., speeding up, slowing down as transmitted by the throttle position sensor or the brake pedal, etc.) and requirements from other parts of the embedded system such as climate control or ESP; the controller acts according to natural factors such as air current temperature, oxygen level, etc. on the one hand, and to environmental annoyances such as exhaust pollution, noise, etc. on the other. It is designed to optimize certain parameters like driving facilities, driving comfort, fuel consumption, etc. One parameter that could be controlled by such a system is the quantity of fuel that has to be injected into each cylinder at each engine cycle according to the engine's revolutions per minute (rpm) and the position of the gas pedal. Another is the ignition timing, and even the so-called variable valve timing (VVT) that controls the time in the engine cycle at which the valves open. There are still others such as the optimal flow of air into the cylinder, the exhaust emission, and the list goes on.

Some information, such as the current rpm, the vehicle speed, etc., are transmitted by this system to another one whose role is to present them to the driver on a dashboard; this last component is actually part of the HMI domain.

The main characteristics for the embedded systems of the power train domain are

- From a functional point of view: The power train control takes into account the different working modes of the motor (slow running, partial load, full load, etc.); this corresponds to various and complex control laws (multivariables) with different sampling periods. Classical sampling periods for signals provided by other systems are l, 2, or 5 ms, while the sampling of signals on the motor itself is in phase with the motor times (from 0.1 to 5 ms).
- From a hardware point of view: This domain requires sensors whose specification has to consider the minimization of the cost/resolution criteria. When it is economically possible for the targeted vehicle, there are also microcontrollers that provide high computation power, thanks to their multiprocessor architecture and dedicated coprocessors (floating point computations), and high storage capacity. Furthermore, the electronic components that are installed into the hardware platform have to be robust to interferences and heat emitted by the engine itself.
- From an implementation point of view: The specified functions are implemented as several tasks with different activation rules according to the sampling rules, with stringent time constraints imposed on task scheduling, mastering safe communications with other systems, and with local sensors/actuators.

Continuous, sampled, and discrete systems are all found in this domain. The control laws contain many calibration parameters (about 2000). Their specification and validation are supported by tools such as Matlab/Simulink [11]. Their deployment and their implementation are the source of a lot of technical problems. For example, underlying control models are generally based on floating point values. If, for economical reasons, the implementation has to be done on a microcontroller without a floating point coprocessor, the programmer has to pay attention to the accuracy of the values in order to be sure to meet the precision required at the specification level of the control laws [12,13]. Another major challenge, as mentioned previously, is to efficiently schedule cyclic activities, because some of them have constant periods, while others have variable periods, according to the motor cycles. This means that scheduling them depends on different logical clocks [14]. Currently, the validation of the control laws is mainly done by simulation and, for their integration, by emulation methods and/or testing. Since the power train domain is subject to hard real-time constraints, performance evaluation and timing analysis activities have to be performed on their implementation models first.

1.2.2 Chassis Domain

The chassis domain is composed of systems whose aim is to control the interaction of the vehicle with the road (wheel, suspension, etc.). Controllers take into account the requests emitted by the driver (steering, braking, or speed up orders), the road profile, and the environmental conditions, like wind, for example. They have to ensure the comfort of the driver and the passengers (suspension) as well as their safety. This domain includes systems like ABS, ESP, automatic stability control (ASC), and four-wheel drive (4WD). The chassis domain is of the utmost importance for the safety of the passengers and of the vehicle itself. Therefore, its development has to be of high quality, as for any critical system.

The characteristics of the chassis domain and the underlying models are similar to those presented for the power train domain: multivariable control laws, different sampling periods, and stringent time constraints (around 10 ms). As for the power train domain, the systems controlling the chassis components are fully distributed onto a networked microcontroller and they communicate with other systems. For example, an ESP system corrects the trajectory of the vehicle by controlling the braking system. Its role is to automatically correct the trajectory of the vehicle as soon as there is understeering or oversteering. To do this, it has to compare the steering request of the driver to the vehicle's response. This is done via several sensors distributed in the vehicle (lateral acceleration, rotation, individual wheel speeds), taking samples 25 times per second. As soon as a correction needs to be applied, it will brake individual front or rear wheels and/or command a reduction of engine power to the power train systems. This system cooperates online with various others such as ABS, electronic damper control (EDC) [15], etc., in order to ensure the safety of the vehicle.

Furthermore, X-by-wire technology, currently applied in avionic systems, is emerging in the automotive industry. X-by-wire is a generic term used when mechanical and/or hydraulic systems are replaced by electronic ones (intelligent devices, networks, computers supporting software components that implement filtering, control, diagnosis, and functionalities). The purpose of such a technology is to assist the driver in different situations in a more flexible way and to decrease production and maintenance cost for braking or steering systems. Nowadays, vehicles equipped with X-by-wire systems have kept traditional mechanical technologies as a backup in case the electronic ones fail. The suppression of this backup presents a major challenge in embedded system design. Conventional mechanical and hydraulic systems have stood the test of time and have proved themselves to be reliable. Therefore, a pure X-by-wire system has to reach at least the same level of safety assessment, with redundancy, replica, functional determinism, and fault tolerance being some of the key underlying words. X-by-wire systems have been used in the avionic industry for some time and so some lessons can be learned from this experience. Nevertheless, due to economical reasons as well as space and weight constraints, the solutions used in an avionics context cannot be compared to that of automotives (in particular, it is impossible to have the same level of hardware redundancy). So, specific fault-tolerant solutions need to be developed. Note that this domain will be mainly concerned by the emerging standard ISO 26262 (committee draft put to the ballot in 2008) on the safety of in-vehicle embedded systems and the certification process that will be required (Section 1.4). It should be noted that, for this domain, the time-triggered software technologies [16,17] bring well-suited solutions despite their lack of flexibility. The Flexray network, the OSEKtime operating system (Offene Systeme und deren schnittstellen für die Elektronik im Kraft-fahrzeug) time operating system and the associated Fault-Tolerant communication (FTCom), or the basic software of AUTomotive Open Standard ARchitecture AUTOSAR (Chapter 2) are good candidates for the implementation of such systems.

1.2.3 Body Domain

The body domain contains functions embedded in a vehicle that are not related to the control of its dynamics. Nowadays, wipers, lights, doors, windows, seats, and mirrors are controlled more and more by software-based systems. In general, they are not subject to stringent performance constraints and, from a safety standpoint, they do not represent a critical part of the system. However, there are certain functions, like an advanced system whose aim is to control access to the vehicle for security, that have to respect hard real-time constraints. It has to be noted that the body functions often involve many communications between each other and consequently have a complex distributed architecture. In this domain emerges the notion of subsystem or subcluster based on low cost sensor–actuator level networks, for example, LIN, which connects modules constructed as integrated mechatronic systems. For example, several functions can be associated to a door: lock/unlock control according to a signal transmitted by a wireless network, window control according to passenger or driver request, as well as mirror control and seat position control. One possible deployment of these functions could be that one main electronic control unit (ECU) supports the reception of the requests (lock/unlock, window up/down, seat up/down, etc.) while the controllers for the motors realizing the requested actions on the physical device (mirror, window, seat) are supported by three other ECUs (Figure 1.2). These four ECUs are connected on a LIN. As some requests concern several doors (e.g., the lock/unlock request), the main ECUs of each door are also connected, for example, on a low-speed CAN. Finally, in order to present the status of the doors to the driver (doors open/close, windows open/close), the information is transmitted by the main ECUs to the dashboard ECU via the CAN low-speed network.

On the other hand, the body domain also contains a central subsystem, termed the central body electronic, whose main functionality is to ensure message transfers between different systems or domains. This system is recognized to be a critical central entity. The body domain functions are related mainly to discrete event applications and their design and validation rely on state machines such as SDL, statecharts, UML state transition diagrams, and synchronous models. These models validate a functional specification, by simulation and, when possible, by model checking. Their implementation, as mentioned before, implies a distribution of this functional specification over hierarchically distributed hardware architecture. High computation power is needed for the central body electronic entity, and, as with the two previous domains, fault tolerance and reliability properties are obligatory for body systems. Although timing constraints are not so stringent as those for the power train and chassis systems, the end-to-end response time between stimuli and response must be evaluated,

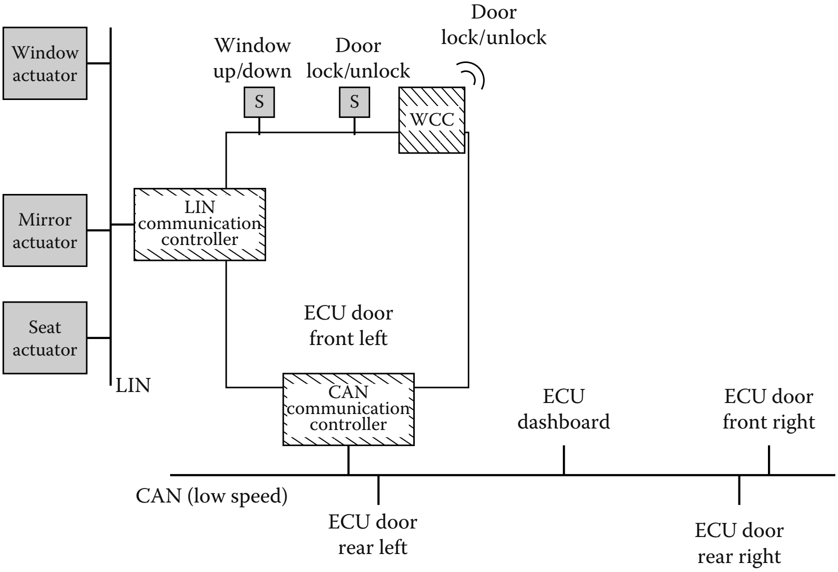


FIGURE 1.2 Example of doors control and of its deployment.

taking into account the performances of the hardware platform, the scheduling policies for each microcontroller, and the network protocol. In fact, the designer has to prove that these response times are always acceptable and therefore that the responses of each stimulus are done in a bounded interval. One challenge in this context is, first, to be able to develop an exhaustive analysis of state transition diagrams and, second, to ensure that the implementation respects the fault tolerance and safety constraints. The problem here is to achieve a balance between the time-triggered approach and flexibility.

1.2.4 Multimedia, Telematic, and HMI

Telematics in vehicles includes systems that support information exchanges between vehicles or between vehicle and road infrastructures. For example, such systems are already used for collecting road tolls; in the near future, telematics will make it possible to optimize road usage through traffic management and congestion avoidance (Chapter 3), to automatically signal road collisions, to provide remote diagnostics (Section 1.2.6), or even to provide access to on-demand navigation, on-demand audio-video entertainment, Web surfing, sending or receiving e-mails, voice calls, short message services (SMSs), etc.

HMI systems support, in a general sense, the interaction between the driver and the passengers with numerous functions embedded in the car. Their main functionalities are, on the one hand, presenting information about the status of the car (e.g., the vehicle speed, the oil level, the status of a door, the status of lights, etc.), the status of a multimedia device (e.g., current frequency for a radio device, etc.), or the result of a request (e.g., visualization of a map provided by a navigation system) and, on the other hand, receiving requests for multimedia equipment (command for radio, navigation systems, etc.). The next generation of multimedia devices will provide new sophisticated HMIs related mainly to entertainment activities. A challenge for HMI system development is thus to take into account, not only the quality, performance, and comfort of the system, but also the impact of this technology on safety [18]. In fact, using HMI must be simple and intuitive, and should not disturb the driver. One way to control the multimedia systems is to avoid too many buttons. The commands should be grouped in a way that minimizes the movements of the driver. For example, the most common solution is to group them on the steering hand wheel—there are as many as 12 buttons on a steering wheel for a high-end model, causing potential confusion between them. Where and how to present information to the driver is also a major problem. The information needs to be clear and should not distract the driver's attention from the road. For example, in the new Citroën C6 a head-up display (HUD) allows key driving information (the vehicle speed, etc.) to be shown on the windscreen in the driver's direct line of vision. Thanks to such a system, the driver can read the information without looking away from the road, as is now done with a traditional dashboard.

Multimedia and telematic devices will be upgradeable in the future and, for this domain, a plug-and-play approach is preferable. These applications need to be portable and the services furnished by the platform (operating system and/or middleware) should offer generic interfaces and downloading facilities. The main challenge here is to preserve the security of the information from, to, or within the vehicle. Sizing and validation do not rely on the same methods as those for the other domains. Here, we shift from considering messages, tasks, and deadline constraints toward fluid data streams, bandwidth sharing, and multimedia quality of service, and from safety and hard real-time constraints toward security for information and soft real-time constraints. Nevertheless, the optimal sizing of these systems can be difficult to determine. For example, a telematics and multimedia platform integrated into a high-end car can be composed of two processors on which about 250 threads running on a Java Machine or on a multitask operating systems will be scheduled. These threads are in charge of handling the data stream and their schedule must give the quality of service required by the user. This kind of system is recognized as "soft" or "firm" real time because it is admissible for some instances of threads, depending on the current load of the processor, to be rejected without significantly reducing the quality of service.

According to experts of this domain, communication between a car and its environment (vehicle-to-vehicle [V2V] or vehicle-to-infrastructure [V2I]) will become more and more important in future years and will bring with it various services with strong added value. The future technologies in this domain begin with efficient voice recognition systems, line-of-sight operated switches, virtual keyboards, etc. but will evolve to include new systems that monitor the status of the vehicle and consequently manage the workload of the driver by avoiding, for example, the display of useless information.

1.2.5 Active/Passive Safety

Demands for vehicles ensuring the safety of driver and passengers are increasing, and are both customer-driven as well as regulatory-based. As mentioned in Section 1.1, the challenge to the automotive industry is to design cars whose embedded systems are able to reach the required safety level at minimal costs. In fact, automotive embedded safety systems target two objectives: "active safety" and "passive safety," the former letting off a warning before a crash and the latter acting after a crash. Seat belts and airbags are examples of systems that help to reduce the effects of an accident, and so they contribute to passive safety. Nowadays, the passive safety domain has reached a good maturity level. An airbag is controlled by a complex algorithm embedded on an ECU and consumes information provided by other systems. Alerted by signals coming from various sensors (deceleration, vehicle speed), this algorithm regulates the right moment to deploy the airbags. The device has to work within a fraction of a second from the time a crash is detected by the sensor to its activating the airbag. As far back as 1984, the U.S. government required cars being produced after April 1, 1989 to have airbags on the driver's side (U.S. Department of Transportation) and in 1998, dual front airbags also became mandatory. Active safety refers to avoiding or minimizing an accident and systems such as braking systems, ABS, ESP, lane keeping, etc., have been specified and marketed for this purpose. The most advanced technological solutions (Chapter 3) are adaptive cruise control and collision warning/avoidance/mitigation systems that contribute to the concept of advanced driver assistance. In general, active safety systems interpret signals provided by various sensors and other systems to assist the driver in controlling the vehicle and interact strongly with almost all the systems embedded in the car.

1.2.6 Diagnostic

As shown in the examples presented in the previous sections, nowadays the complexity of electronic architectures embedded in a car infers functions deployed on several microcontrollers to intensively interact between themselves. Therefore, diagnosis has become a vital function throughout the lifetime of a vehicle. So, any system that can help to access and relate information about a car is obviously very important and should be designed simultaneously with the original design of the car. In particular, specifying a system that is able to collect information and establish onboard diagnostics (OBD) is advantageous for the vehicle's owner as well as for a repair technician. The generic term used for this function is "onboard diagnostics" or OBD. More precisely, this concept refers to self-diagnosis and reporting facilities, which were made possible with the introduction of computer-based systems that could memorize large amounts of information. While the role of early diagnostic functions was limited to a light switching on as soon as a specific problem was detected, recent OBD systems are based on standardized communication means—a standardization of monitored data and a standardized coding and reporting of a list of specific failures, termed diagnostic trouble codes (DTC). Thanks to this standardization effort, the memorized values of parameters can be analyzed through a single compliant device. The underlying intent to this standardization effort was a regulatory constraint on exhaust emission control systems throughout the useful lifetime of a vehicle. The OBD-II specification, mandatory for all cars sold in the United States as of 1996, precisely defines the diagnostic connector, the electrical signaling protocol, the messaging format, as well as the vehicle parameters that can be monitored. In 2001, the European Emission Standards Directive 98/69/EC [19] established the requirement of EOBD, a variant of OBD-II, for all petrol vehicles sold in the European Union as of January 2001. Several standards have been successively provided: ISO 9141-2 concerns a low-speed protocol close to that of RS-232 [20], ISO 14230 introduced the protocol KWP2000 (Keyword Protocol 2000) that enables larger messages [21], and ISO 15765 proposes a diagnostic, termed Diag-on-CAN, which uses a CAN [22]. The next step forecast will enable reporting emissions violations by the means of a radio transmitter.

1.3 Standardized Components, Models, and Processes

As pointed out in Section 1.1, the design of new in-vehicle embedded systems is based on a cooperative development process. Therefore, it must ensure the interoperability between components developed by different partners and ease their portability onto various platforms in order to increase the system's flexibility. On the one hand, a means to reach these objectives is furnished by the standardization of services sharing hardware resources between application processes, in particular, networks and their protocols and operating systems. On the other hand, portability is achieved through the specification of a common middleware. Notice that such a middleware also has to deal with the interoperability properties. Finally, a standardized and common support for modeling and documenting systems all along their development eases the design process itself and, more specifically, the exchanges between the different partners at each step of the development cycle. In the following, we introduce some of the standardized components or models aiming to support this cooperative development process.

1.3.1 In-Vehicle Networks and Protocols

Specific communication protocol and networks have been developed to fulfill the needs of automotive-embedded systems. In 1993, the SAE Vehicle Network for Multiplexing and Data Communications Standards Committee identified three kinds of communication protocols for in-vehicle embedded systems based on network speed and functions [23]; they are called, respectively, "class A," "class B," and "class C." The same committee also published a requirement list concerning safety critical applications. In particular, the communication protocol for X-by-wire systems must respect requirements for "dependability and fault-tolerance" as defined for class C [24]. Networks compliant to class A provide a bit rate below 10 kbps and are dedicated to sensor and actuator networks; the LIN bus and TTP/A bus are among the most important protocols in this class. Class B specifies a medium speed (10–500 kbps) and is thus convenient for transferring information in vehicle-centric domains and the body's electronics systems. CAN-B is a widely used class B protocol. Class C has been defined for safety-relevant systems in power train or chassis domains. The data rates here are lower than 1 Mbps. CAN-C (high-speed CAN), TTP/C, and FlexRay fall into this category. They have to provide highly reliable and fault tolerant communication. Obviously, class C networks will be required in future X-by-wire applications for steering and braking. For further information on automotive-embedded networks, the reader can refer to Chapter 4 as well as to Refs. [25,26].

1.3.2 Operating Systems

OSEK/VDX [27] is a multitask operating system that is becoming a standard in the European automotive industry. This standard is divided in four parts: OSEK/VDX OS is the specification of the kernel itself; OSEK/VDX COM concerns the communication between tasks (internal or external to an ECU); OSEK/VDX NM addresses network management; and finally, OSEK/VDX OIL is the language that supports the description of all the components of an application. Certain OSEK-targeted applications are subject to hard real-time constraints, so the application objects supported by OSEK have to be configured statically.

OSEK/VDX OS provides services on objects like tasks ("basic tasks," without blocking point, and "extended tasks," that can include blocking points), events, resources, and alarms. It proposes a fixed priority (FP) scheduling policy that is applied to tasks that can be preemptive or non-preemptive, and combined with a reduced version of the priority ceiling protocol (PCP) [28,29] in order to avoid priority inversion or deadlock due to exclusive resource access. Intertask synchronization is achieved through private events and alarms. The implementation of an OSEK/VDX specification has to be compliant to one of the four conformance classes—BCC1, BCC2, ECCI, ECC2—that are specified according to the supported tasks (basic only or basic and extended), the number of tasks on each priority level (only one or possibly several), and the constraints of the reactivation counter (only one or possibly several). BCC1 defines a restricted implementation that aims to minimize the size of the corresponding memory footprint, the size of the data structures, and the complexity of the management algorithms. ECC2 specifies the implementation of all the services. The MODISTARC project (Methods and tools for the validation of OSEK/VDX based DISTributed ARChitectures) [30] provided the relevant test methods and tools to assess the compliance of OSEK/VDX implementations.

In order to describe an application configuration, the OSEK consortium provided a specific language, called OSEK/VDX OIL (OSEK Implementation Language). This language allows, for one ECU, the description of several application configurations, called application modes. For example, the application configurations can be specified for a normal operation mode, for a diagnosis mode, and for a download mode.

The dependability purpose and fault tolerance for critical applications is usually achieved by a time-triggered approach [17]. So, the time-triggered operating system OSEKtime [27] was defined. It supports static and time-triggered scheduling, and offers interrupt handling, dispatching, system time and clock synchronization, local message handling, and error detection mechanisms. Thanks to these services, an application running on OSEKtime can be predictable. OSEKtime is compatible to OSEK/VDX and is completed by FTCom layer for communication services.

It should be noted that the specification of the basic software for AUTOSAR (Chapter 2) is based on services from OSEK and OSEKtime. Commercial implementations of OSEK/VDX standard are available [27] and open-source versions as well [31,32].

Rubus is another operating system tailored for the automotive industry and used by Volvo Construction Equipment. It was developed by Arcticus systems [33]. Rubus OS is composed of three parts: the Red Kernel, which manages the execution of off-line scheduled time-triggered tasks; the Blue Kernel, which is dedicated to the execution of event-triggered tasks; and the Green Kernel, which is in charge of external interrupts. As for OSEK/VDX OS, the configuration of the tasks has to be defined statically off-line.

For multimedia and telematics applications, the operating systems are generic ones, such as VxWorks (from WindRiver) or even a Java machine. "Microsoft Windows Automotive 5.0" extends the classical operating system Windows CE with telematic-oriented features and was, for example, installed among others in certain Citroën Xsara and the BMW 7 series.

1.3.3 Middleware

Flexibility and portability of applicative components require two main features. On the one hand, an application embedded on a distributed platform is based on the description of elements, the semantics of the interaction types among the elements, and, consequently, the semantics of their composition. Note that these interactions must be specified disregarding the allocation of components on an ECU. On the other hand, the properties required at the application level, mainly timing and dependability properties, must be met when components are allocated onto a technical platform (operating systems, communication drivers and protocol, input/output [I/O] drivers, etc.). Traditionally, these features are achieved through the specification of a middleware. Firstly, the structure of the middleware, that is, the elementary software components allocated on each ECU and the way they interact, has to be formally identified and, secondly, the interface services that furnish a way for applicative components to use the middleware services independently of their allocation have to be furnished. During the last decade, several projects focused on this purpose (see, e.g., the German Titus project [34] started by DaimlerChrysler in 1994). The purpose of this project was to develop an interface-based approach similar to the ROOM methodology [35], but differing considerably in certain details, mainly in making an "actor-oriented" approach that was suitable for ECU software. The French EEA project [36] identified the classes of software components implemented on an ECU. Then the European ITEA EAST EEA project refined these classes and proposed a more advanced architectural view of an ECU [10]. The mission of the DECOS project, supported by the sixth EU Framework Program, was to develop an architecture-based design methodology, to identify and specify the associated components off-the-shelf (COTS) hardware and software components, and to provide certified development tools and advanced hybrid control technologies. This project targeted control systems concerning the dependability of software-intensive systems, in particular, in avionics (airbus) and automotive industries. After that, the Volcano project concentrated on just the communication services and provided a way (both middleware components and interface services) for supporting the signal exchanges between distant applicative components by hiding the underlying protocol. Volcano targeted the timing properties imposed on signal exchanges [37,38].

Finally, the AUTOSAR consortium (see Chapter 2 for more details) standardized a common software infrastructure for automotive systems [39]. Once put into practice, it will bring meaningful progress when designing an embedded electronic architecture because (1) it will allow the portability of the functions on the architecture and their reuse, (2) it will allow the use of hardware COTS, and (3) on a same ECU it will be able to integrate functions from different suppliers. During the lifetime of the car, this standard will facilitate updating the embedded software as this technology evolves, as well as the maintenance for the computers.

1.3.4 Architecture Description Languages for Automotive Applications

Sharing the same modeling language between the different partners involved in the design of these in-vehicle embedded systems is a means to support an efficient collaborative development process. In this context, such a language will have to allow for describing a system at different steps of its development (requirement specification, functional specification, design, implementation, tuning, etc.) by taking into consideration the different viewpoints of the actors as well as ensuring a consistency between these different views. It will also need to reflect the structure of the embedded systems as an architecture of components (hardware components, functional components, software components). The concept of architecture description languages (ADLs), developed for large software applications [40], is well suited to these objectives. ADLs are used to describe the structure of a system by means of the components that are interconnected in a way to form configurations. These descriptions are free of implementation details, one of the objectives being the mastery of the structure of complex systems. Thus the composition (associated to hierarchy) used to specify the assembly of the elements constitutes the fundamental construction. For critical systems, as is the case in automotive electronics, an ADL must support not only the specification of the functional aspects of the system, but also those that are extra-functional (timing properties, dependability properties, safety properties), and other transformation and verification facilities between design and implementation, while maintaining a consistency between the different models. In 1991, Honeywell Labs specified an ADL, MetaH [41], that was dedicated to avionics systems. This language was chosen in 2001 to be the core of an avionics ADL (AADL) standard under the SAE authority [42]. For the specific automotive domain, several languages were proposed (Chapter 9). For example, the language EAST-ADL [43], which is tightly related to the generic reference architecture mentioned in the previous section, was specified in the European ITEA EAST-EEA project [10] and extended in the ATESST project [44]. The purpose of EAST-ADL is to provide support for the nonambiguous description of in-car embedded electronic systems at each level of their development. It provides a framework for modeling such systems through five abstraction levels, divided into seven layers (also called artifacts), as shown in Figure 1.3. Some of these layers are mainly concerned with software development while others are linked to the execution

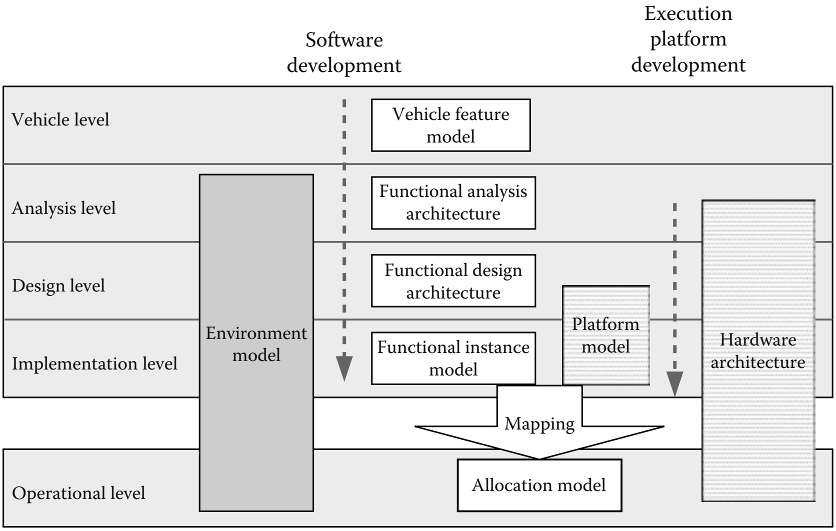


FIGURE 1.3 The abstraction levels and the system views in EAST-ADL.

platform (ECUs, networks, operating systems, I/O drivers, middleware, etc.). All these layers are tightly linked, allowing traceability among the different entities that are implicated in the development process. Besides the structural decomposition, which is typical for any software development or modeling approach, the EAST-ADL also has means for modeling cross-cutting concerns such as requirements, behavioral description and validation, and verification activities. At vehicle level, the vehicle feature model describes the set of user-visible features. Examples of such features are antilock braking or windscreen wipers. The functional analysis architecture, at the analysis level, is an artifact that represents the functions that realize the features, their behavior, and their cooperation. There is an n-to-n mapping between vehicle feature model and functional analysis architecture entities, that is, one or several functions may realize one or several features. The functional design architecture (design level) models a decomposition or refinement of the functions described at analysis level in order to meet constraints regarding allocation, efficiency, reuse, supplier concerns, etc. Again, there is an n-to-n mapping between the entities for functional design architecture and the corresponding ones in functional analysis architecture. At the implementation level, the role of the function instance model is to prepare the allocation of software components and exchanged signals to OS tasks and frames. It is, in fact, a flat software structure where the functional design architecture entities have been instantiated. It provides an abstraction of the software components to implement. In order to model the implementation of a system, EAST-ADL furnishes, on the one hand, a way to describe the hardware platforms and their available services (operating system, protocol, middleware) and, on the other hand, a support for the specification of how a function instance model is distributed onto a platform. This is done thanks to three other artifacts. The hardware architecture includes the description of the ECUs and, more precisely, those for the microcontroller used, the sensors and actuators, the communication links (serial links, networks), and their connections. The platform model defines the operating system and/or middleware application programming interface (API) and, in particular, the services provided (schedulers, frame packing, memory management, I/O drivers, diagnosis software, download software, etc.). Finally, the allocation model is used at the operational level. It models the tasks that are managed by the operating systems and frames, which are in turn managed by the protocol. This is the result of the function instance model entities being mapped onto the platform model. Note that the specification of a hardware architecture and a platform model is done simultaneously with function and software specification and can even be achieved during the definition of an allocation model. At this lowest abstraction level, all of the implementation details are captured. The EAST-ADL language provides consistency within and between the artifacts belonging to the different levels from a syntactic and semantic point of view. This makes an EAST-ADL-based model a strong and nonambiguous support, not only for the realization of software components, but also for building, automatically, models that are suited for format validation and verification activities [45,46].

1.4 Certification Issue of Safety-Critical In-Vehicle Embedded Systems

Several domains are recognized as critical, for example, nuclear plants, railways, avionics. They are subject to strong regulations and must prove that they meet rigorous safety requirements. Therefore, the manner of specification and the management of the dependability/safety properties represent an important issue, as well as the certification process. This problem has become of primary importance for the automotive industry due to the increasing number of computer-based systems such as critical functions like steering and braking. Consequently, several proposals have been under study. The existing certification standards [47], ARP 4754 [48], RTCA/DO-178B [49] (used in avionics), or EN 50128 [50] (applied in the railway industry), provide stringent guidelines for the development of a safety-critical embedded system. However, these standards are hardly transposable for in-vehicle software-based systems: partitioning of software (critical/noncritical), multiple versions, dissimilar software components, use of active redundancy, and hardware redundancy. In the automotive sector, the Motor Industry Software Reliability Association (MISRA), a consortium of the major actors for automotive products in the United Kingdom, proposes a loose model for the safety-directed development of vehicles with onboard software [51]. Also, the generic standard IEC 61508 [52], used for electrical/electronic/programmable electronic systems appears to be a good candidate for supporting a certification process in the automotive industry. Finally, an upcoming standard is being developed, derived from that for the IEC, which serves automotive-specific needs.

The ISO international draft standard ISO WD 26262, planned for 2008, is currently under progress in cooperation with the EU, the United States, and Japan [53,54]. The next step will consist in the assessment of its usability by the members of the ISO association. The ISO WD 26262 standard is applied to functional safety, whose purpose is to minimize the danger that could be caused by a possibly faulty system. The ISO draft specifies that functional safety is ensured when "... a vehicle function does not cause any intolerable endangering states, which are resulting from specification, implementation or realization errors, failure during operation period, reasonably foreseeable operational errors [and/or] reasonably foreseeable misuse." This definition concerns, in fact, the entire life cycle of a system. Safety control has to be effective during the preliminary phase of the system design (in particular, hazard analysis and risk assessment), during development (functional safety requirement allocation for hardware and software, and system evaluation), and even during operation services and decommissioning (verification that assumptions made during safety assessment and hazard analysis are still present). Once the function of a system has been specified, the safety process dictates that it goes over an established list of driving situations and their corresponding malfunctions and, for each one of them, gives the safety functions that are specified to avoid such situations as well as how to maintain the vehicle in a safe mode. Each of these situations is characterized by the frequency of its occurrences, the severity of the damage, and the controllability of the situation by a driver. The system is characterized according to these parameters by a so-called automotive safety integrity level (ASIL). The format definition of the safety properties associated to each ASIL is not known at the present time. If we refer to the generic standard IEC 61508 [52], each SIL is defined by two kinds of safety properties: functional requirements, that is, no erroneous signals are produced by an ECU, and safety integrity attributes, that is, the probability of dangerous failure occurrences per hour has to be less than a given threshold (e.g., less than 10^{-8}). Throughout the development of the system that realizes a function, it must be verified that this system ensures all the properties required by the SIL assigned to the function. Verification activities are based, for example, on failure mode and effect analysis (FMEA), fault or event tree analysis, etc. completed by several techniques that could depend on the development process stage (formal methods and model checking, performance evaluation, schedulability and timing analysis, probability, hardware in the loop, system in the loop, etc.).

1.5 Conclusion

Nowadays, for any activity in our everyday life, we are likely to use products or services whose behavior is governed by computer-based systems, also called embedded systems. This evolution also affects the automotive industry. Several computers are embedded in today's vehicles and ensure functions that are vehicle centric, such as motor control, braking assistance, etc., as well as passenger centric such as entertainment, seat control, etc. This chapter has shown why this evolution is inescapable and has outlined the main thrusts of this development. First, state regulations, such as controlling exhaust emissions or mandatory active safety equipments (e.g., airbags), which impose embedding complex control laws that can only be achieved with computer-based systems. Second, customers are asking for more comfortable, easy-to-drive, and safe cars and carmakers are aiming to launch new innovative products; both are costly. Today's advancing software technology appears to be a good trade-off between cost and product development, and therefore facilitates the introduction of new services in cars. In order to identify the requirements applied to embedded electronic systems, we presented a classification of these systems according to well-identified functional domains. The pressure of these requirements affects the technological solutions in terms of hardware components as well as software development. Finally, the economical constraints push for the emergence of standards easing hardware/software independence, and consequently an efficient collaborative development process of embedded electronic architectures (Chapter 2) and the reuse of hardware and software entities (Chapter 8). For example, at the present time, the CAN is predominant in the interconnection of the ECUs. However, due to the increase in exchanges between ECUs, other solutions are emerging (e.g., the FlexRay network, the integration of mechatronic systems deployed on hierarchical distributed architecture, etc.). The growing complexity of the software embedded in a car reflects a well-mastered development process. Autonomous and automated road vehicles, communicating cars, and integrated traffic solutions are keywords for the vehicle of the future. These trends target controlling motorized traffic, decreasing congestion and pollution, and increasing safety and quality of lives (Chapter 3). In such a scenario, the development of a vehicle cannot be considered separately, but must be seen as part of a complex system. Furthermore, the next standard OSI 26262, and those that are already being applied for road traffic, form another strong argument for solid, structured design methods. Thanks to international initiatives, such as AUTOSAR, the concepts of model-based development (MBD), model-driven development (MDD), and component-based software engineering (CBSE) are penetrating the culture of automotive system designers. This will be possible as soon as tools supporting these concepts, and suited to the automotive industry, reach a higher level of maturity.

References

1. OICA. International Organization of Motor Vehicle Manufacturers. http://www.oica.net.
2. SAE. International Society of Automotive Engineers. http://automobile. sae.org/.
3. P. Hansen. New S-class Mercedes: Pioneering electronics. *The Hansen Report on Automotive Electronics*, 18(8), October 2005.
4. J. Leohold. Communication requirements for automotive systems. In *Slides Presented at the 5th IEEE International Workshop on Factory Communication System*, WFCS'2004, Vienna, Austria, September 2004.
5. G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1), January 2002, pp. 88–93.
6. E. Knippel and A. Schulz. Lessons learned from implementing configuration management within electrical/electronic development of an automotive OEM. In *International Council on Systems Engineering, INCOSE 2004*, Toulouse, France, June 2004.
7. S. Fürst. AUTOSAR for safety-related systems: Objectives, approach and status. In *Second IEE Conference on Automotive Electronics*, London, United Kingdom, March 2006.
8. A. Sangiovanni-Vincentelli. Automotive electronics: Trends and challenges. In *Convergence 2000*, Detroit, MI, October 2000.

9. F. Simonot-Lion. In-car embedded electronic architectures: How to ensure their safety. In *Fifth IFAC International Conference on Fieldbus Systems and their Applications, FeT'2003*, Aveiro, Portugal, July 2003.
10. ITEA EAST-EEA project. http://www.east-eea.net.
11. The MathWorks. MATLAB/SIMULINK. http://www.mathworks.com.
12. T. Hilaire, Ph. Chevrel, and Y. Trinquet. Designing low parametric sensitivity FWL realizations of LTI controllers/filters within the implicit state-space framework. In *44th IEEE Conference on Decision and Control and European Control Conference ECC 2005*, Séville, Spain, December 2005.
13. T. Hilaire, Ph. Chevrel, and J. Whidborne. A unifying framework for finite wordlength realizations. *IEEE Transactions on Circuits and Systems-I, Fundamental Theory and Applications*, 54(8): 1765–1774, 2007.
14. C. André, F. Mallet, and M.-A. Peraldi Frati. A multiform time approach to real-time system modelling. In *IEEE Second International Symposium on Industrial Embedded Systems—SIES'2007*, Lisbon, Portugal, July 2007.
15. A. Schedl. Goals and architecture of FlexRay at BMW. In *Slides Presented at the Vector FlexRay Symposium*, March 2007.
16. TTTech Computertechnik AG. http://www.tttech.com.
17. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395 of *The Springer International Series in Engineering and Computer Science*. Springer, 1997.
18. D. McClure. The HMI challenge—balancing safety with functionality. Technical Report 37, SBD, September 2006.
19. EOBD. Directive 98/69/EC of the European parliament and of the council. *Official Journal of the European Communities*, October 1998.
20. ISO 9141. *Road Vehicles—Diagnostic Systems*. International Organization for Standardization, 1989.
21. ISO 14230. *Road Vehicles—Diagnostic Systems—Keyword Protocol 2000*. International Organization for Standardization, 1999.
22. ISO 15765. *Road Vehicles—Diagnostics on Controller Area Networks (CAN)*. International Organization for Standardization, 2004.
23. Society of Automotive Engineers. J2056/1 Class C Applications Requirements Classifications. In *SAE Handbook*, Vol. 1, 1994.
24. B. Hedenetz and R. Belschner. Brake-by-wire without mechanical backup by using a TTP-Communication Network. Technical report, SAE—Society of Automotive Engineers, Detroit, MI, 1998.
25. N. Navet, Y.-Q. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems, special issue on industrial communications systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.
26. Society of Automotive Engineers. J2056/2 Survey of Known Protocols. In *SAE Handbook*, Vol. 2, 1994.
27. OSEK-VDX. http://www.osek-vdx.org.
28. J. Liu. *Real-Time Systems*. Prentice Hall, Englewood Cliffs, NJ, 1997.
29. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computer*, 39(9):1175–1185, 1990.
30. MODISTARC. http://www.osek-vdx.org.

31. OpenOSEK. http://www.openosek.org/.
32. J.L. Bechennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline: An open source implementation of the OSEK/VDX RTOS. In *11th IEEE International Conference on Emerging Technologies and Factory Automation—ETFA'06*, Prague, September 2006.
33. RUBUS. Arcticus Systems AB. http://www.arcticus.se.
34. U. Freund and A. Burst. Model-based design of ECU software: A component-based approach. In *OMER LNI, Lecture Notes of Informatics, GI Series*, October 2002.
35. B. Selic, G. Gullekson, and P.T. Ward. *ROOM: Real-Time Object Oriented Modeling*. John Wiley, New York, 1994.
36. F. Simonot-Lion and J.P. Elloy. An architecture description language for in-vehicle embedded system development. In *15th Triennial World Congress of the International Federation of Automatic Control—B'02*, Barcelona, Spain, July 2002.
37. A. Rajnak. *Volcano—Enabling Correctness by Design*. CRC Press, Taylor & Francis, Boca Raton, FL, 2005.
38. A. Rajnak, K. Tindell, and L. Casparsson. Volcano communications concept—Technical report. Technical report, Volcano Communications Technologies AB, 1998.
39. H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürstand, K.P. Schnelle, W. Grote, N. Maldenerand, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Voget, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and exploitation of the AUTOSAR development partnership. In *Convergence 2006*, Detroit, MI, October 2006.
40. N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
41. S. Vestal. Metah support for real-time multi-processor avionics. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS '97*, Geneva, Swiss, April 1997. IEEE Computer Society.
42. P.H. Feiler, B. Lewis, and S. Vestal. The SAE avionics architecture description language (AADL) standard. In *9th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2003*, Washington, DC, May 2003.
43. U. Freund, O. Gurrieri, J. Küster, H. Lönn, J. Migge, M.O. Reiser, T. Wierczoch, and M. Weber. An architecture description language for developing automotive ECU-software. In *International Conference on Systems Engineering, INCOSE 2004*, Toulouse, France, June 2004.
44. ATESST. Advancing traffic efficiency and safety through software technology. EC IST FP6 project, http://www.atesst.org.
45. V. Debruyne, F. Simonot Lion, and Y. Trinquet. EAST-ADL—an architecture description language—validation and verification aspects. In P. Dissaux, M. Filali, P. Michel, and F. Vernadat, editors, *Architecture Description Language*. Kluwer Academic Publishers, 2004, pp. 181–196.
46. V. Debruyne, F. Simonot-Lion, and Y. Trinquet. EAST ADL, an architecture description language—validation and verification aspects. In *IFIP World Computer Congress 2004—Workshop on Architecture Description Languages*, Vol. 176 of *IFIP Book Series*, Toulouse, France, September 2004. Springer, pp. 181–196.

47. Y. Papadopoulos and J.A. McDermid. The potential for a generic approach to certification of safety-critical systems in the transportation sector. *Journal of Reliability Engineering and System Safety*, 63:47–66, 1999.
48. SAE International. Certification Considerations for Highly-Integrated or Complex Aircraft Systems. International Standard, November 1996.
49. RTCA DO-178B. Software Considerations in Airbone Systems and Equipment Certification. International Standard. Radio Technical Commission for Aeronautics, 1994.
50. EN50128. *Railway Applications—Software for Railway Control and Protection Systems*. International Standard, CENELEC, 2001.
51. P.H. Jesty, K.M. Hobley, R. Evans, and I. Kendall. Safety Analysis of Vehicle-Based Systems. In *Eighth Safety-Critical Systems Symposium*, Southampton, United Kingdom, 2000. Springer.
52. IEC. IEC 61508-1, Functional Safety of Electrical/Electronic/ Programmable Safety-related Systems—Part 1: General Requirements, IEC/SC65A. International Standard, 1998.
53. ISO WD 26262. Automotive Standards Committee of the German Institute for Standardization: Road Vehicles—Functional Safety. Preparatory Working Draft.
54. M. Findeis and I. Pabst. Functional safety in the automotive industry, process and methods. In *VDA Alternative Refrigerant Winter Meeting*, Saalfelden, Austria, February 2006.

2

Application of the AUTOSAR Standard

Stefan Voget
Continental Automotive GmbH

Michael Golm
Siemens AG

Bernard Sanchez
Continental Automotive GmbH

Friedhelm Stappert
Continental Automotive GmbH

2.1 Motivation

Today, development of electronic control units (ECUs) is characterized by several driving factors:

- Demands for more services, security, economy, and comfort
- Increasing complexity due to more ECUs and the growth in sharing software and functionality [1–5]
- More diversity of ECU hardware and networks (controller area network [CAN], local interconnect network [LIN], FlexRay, MOST, etc.)

As a result of these driving factors, communication between ECUs is increasing. Unfortunately, the ECU networks are oriented for a distribution of automotive functions which, despite the fact that it has been evolving for some time, is still not structured with respect to the newest technologies. The ECUs are grouped into several subdomains (Chapter 1), for example, power train, body, telematic, chassis, etc. Before AUTomotive Open System ARchitecture (AUTOSAR), the networks of ECUs were neither standardized in accordance with their interfaces across these subdomain borders nor developed with respect to the interrelationships between the nodes of the network.

Comparable statements can be made for the development processes. The software development processes for different ECUs evolved according to the individual history of the subdomains, and were quite divergent for a long time. In the automotive industry, most of the widespread system development processes assign functional requirements to software and hardware components on a one-to-one basis.

2.1.1 Shortcomings in Former Software Structures

The increasing total share of software resulted in high complexity and high costs. This became more critical with nonstandardized development processes and inadequate networks. In addition, the incorporation of third-party software made the collaboration between companies even more complex.

An appropriate level of abstraction in the software architecture modeling and appropriate integration concepts were still missing. The architectures did not reflect the effects of quality requirements. As a consequence, these often remained vague and unexplored. The architectures evolving with a single solution development strategy did not represent long-term solutions.

To further complicate matters, a lot of the functionalities are distributed over several ECUs, for example, the software that controls the lights of the indicator functionality is distributed over up to eight ECUs in high-end vehicles. Moreover, some of the future functionalities are not realizable with a loose side-by-side of the ECUs, for example, drive-by-wire will need a very close and safe interlocking of ECUs across different domains [6]. The traditional split of automotive functions will require more and more interconnectivity with the new upcoming functionalities.

2.1.2 Setting up AUTOSAR

With respect to this background, the leading automobile companies and their first-tier suppliers formed a partnership in 2003. This partnership has established an industry-wide standard for the automobile electronic, AUTOSAR, which is headed by the following 10 "core partners": BMW Group, Bosch, Continental, DaimlerChrysler, Ford Motor Company, General Motors, PSA Peugeot Citroën, Siemens VDO, Toyota Motor Corporation, and Volkswagen. The first phase of AUTOSAR started in 2003 and ended in 2006. During this phase, 52 "premium members," companies of the suppliers, and software and semiconductor industries joined the development of this standard and made major contributions in the consortium. In addition to these premium members, there are "associate members," "development members," and "attendees," whose roles in AUTOSAR varied with respect to their contribution and exploitation (www.autosar.org).

The first phase of AUTOSAR finished at the end of 2006, and the first AUTOSAR products were made available on the market.

The members of AUTOSAR agree that AUTOSAR makes it possible to control the complexity of the electrical and electronic components, together with an increase in quality and profitability. The future of automotive engineering is in these modular and scalable AUTOSAR software architectures.

2.1.3 Main Objectives of AUTOSAR

The main principle of AUTOSAR is "cooperate on standards, compete on implementation." Thus, the members established a set of main objectives, which also needed to be standardized because they were not considered primary factors for competitiveness [7–10].

- Consideration of availability and safety requirements
- Redundancy activation
- Scalability to different vehicle and platform variants
- Implementation and standardization of basic functions as an industry-wide "standard core" solution
- Transferability of functions from one ECU to another within the network of ECUs
- Integration of functional modules from multiple suppliers
- Maintainability throughout the whole product life cycle
- Increased use of commercial-off-the-shelf (COTS) hardware
- Software updates and upgrades over vehicle lifetime

On all levels of modeling, an object-oriented eXtensible Markup Language (XML) class model, specified in UML 2.0, is used. Starting with the explanation of the description language you have to go down (via the metamodel-level) to a concrete user model, which can be realized as a concrete XML file.

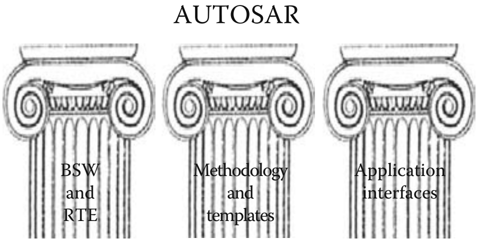


FIGURE 2.1 The three pillars of AUTOSAR.

The most important consequence of the stringent component-based approach of AUTOSAR, concerning the development process, is a separation of application development from the lower levels of the integration development (basic software [BSW]). The separator between these two parts is the AUTOSAR runtime environment (RTE), which concretely realizes the concept of a virtual functional bus (VFB) as an abstracting communication principle. The idea of this concept is that an application does not need to know the concrete paths from data and signals below RTE when two applications communicate together.

This simplifies matters for the application developer. He or she, de facto, needs no further knowledge about concrete architectures, even if a deeper knowledge about the interfaces, made available to an application on top of the RTE, is still indispensable (Figure 2.1).

2.1.4 Working Methods in AUTOSAR

AUTOSAR was set up as a partnership to define an industry-wide standard. The consortium tries to use as many existing solutions as possible, trying not to invent everything newly. In most cases, standardization means choosing one option over several alternatives. Of course, as different solutions are already implemented by the companies, this means agreeing on compromises. If possible, existing standards are taken as they are, for example, CAN, LIN, OSEK, etc. In other cases, if it is not possible to agree directly on an existing solution, cooperation with other standardization groups is established, for example, with FlexRay consortium, ASAM-FIBEX,* MOST, etc.

To be able to standardize one needs a minimal stability and some common understanding of the issue one is dealing with. Therefore, AUTOSAR was not started without any preparation. Several research projects were carried out in advance, in particular, the projects ITEA-EAST/EEA† and AEE,‡ which can be mentioned here.

* Field bus exchange format from Association for Standardization of Automation and Measuring Systems and Measuring Systems.

† An European-funded project for embedded electronic architecture (EEA).

‡ A French funding project; a predecessor of ITEA-EST/EEA.

2.2 Mainstay of AUTOSAR: AUTOSAR Architecture

2.2.1 AUTOSAR Concept

To fulfill the requirements discussed in the previous chapter and in Ref. [11], the AUTOSAR consortium defined a new development methodology for automotive software and software architecture. The development methodology is focused on a model-driven development style. The software architecture, as well as the ECU hardware and the network topology, are modeled in a formal way, which is defined in a metamodel that supports the software development process from architecture up to integration. All available modeling elements are specified by the "AUTOSAR metamodel" [12]. The metamodel is defined according to the rules of the OMG Meta Object Facility [13].

The envisioned development methodology starts by defining the software architecture. An exemplary software architecture can be seen in Figure 2.2.

The boxes represent software "components." At the perimeter of the boxes the communication "ports" of the software components are shown. A port with an inward pointing triangle is a "required port." A port with an outward pointing triangle is a "provided port." Required ports are the data receivers in a data flow-oriented communication, whereas provided ports are the senders. A provided port can be connected with one or more required ports of other software components. To be able to connect ports, the interfaces of the two ports must be compatible.

There are two types of interfaces, "sender/receiver interfaces" and "client/server interfaces." A sender/receiver interface supports message-based communication, while a client/server interface supports a remote-procedure-call style of communication.

A sender/receiver interface consists of a list of "data elements." Each data element has a name and a data type.

The client/server interface consists of a list of "operations." Each operation has a signature, consisting of a name and a list of "parameters." Each parameter is described

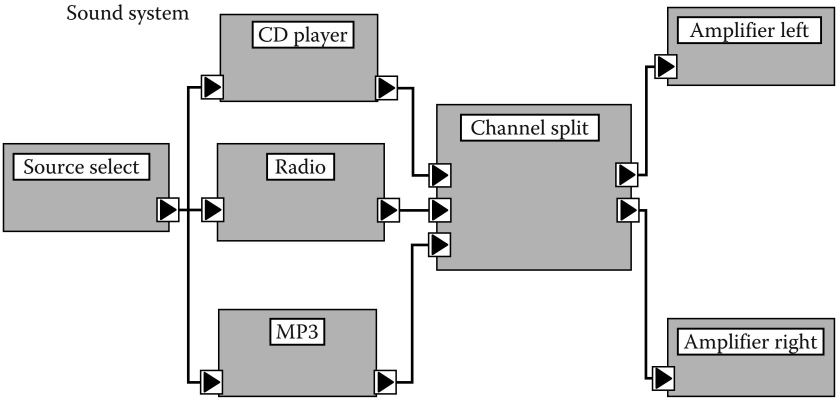


FIGURE 2.2 Example for software components and connectors.

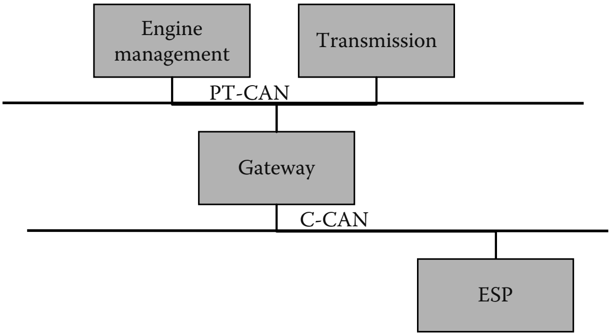


FIGURE 2.3 Examplary network topology.

by a name, a type, and a direction, which can be either in, out, or in–out. The details of all software components related to modeling elements are described in Ref. [14].

The software architecture is defined without consideration of the hardware on which the software components will run on later. This means that two software components might run on the same ECU or on different ECUs. The communication between the components is then either an intra-ECU communication or an inter-ECU communication. To abstract from this difference, AUTOSAR introduces the VFB. The VFB can be seen as a software bus to which all components are attached. The VFB software bus is based on ideas similar to common object request broker architecture (CORBA) [15].

The hardware architecture is modeled in parallel to the definition of the software architecture. AUTOSAR allows for modeling the topology of a vehicle network as well as the hardware of an ECU. An example of this topology can be seen in Figure 2.3.

The example shows two ECUs connected to a power train CAN (PT-CAN) and one ECU connected to a chassis CAN (C-CAN). The two CAN busses are connected through a gateway.

Once the software architecture and the network topology are defined, the software entities can be mapped to the hardware entities. The software component template standardizes the format for describing the software entities and is a very important part of the AUTOSAR metamodel. It defines how the software architecture is specified.

2.2.2 Layered Software Architecture

AUTOSAR defines a software architecture for ECUs. This architecture is defined in a layered style. The lowest layer of the architecture, the microcontroller abstraction layer (MCAL), is responsible for providing abstractions of typical devices. The MCAL modules could be considered as device drivers. There are four groups of MCAL modules: microcontroller drivers, memory drivers, communication drivers, and input/output (I/O) drivers. The communication drivers include drivers for CAN, LIN, and FlexRay. The I/O drivers include drivers for pulse width modulation (PWM), analog-to-digital converter (ADC), and digital I/O (DIO). Above the MCAL there is

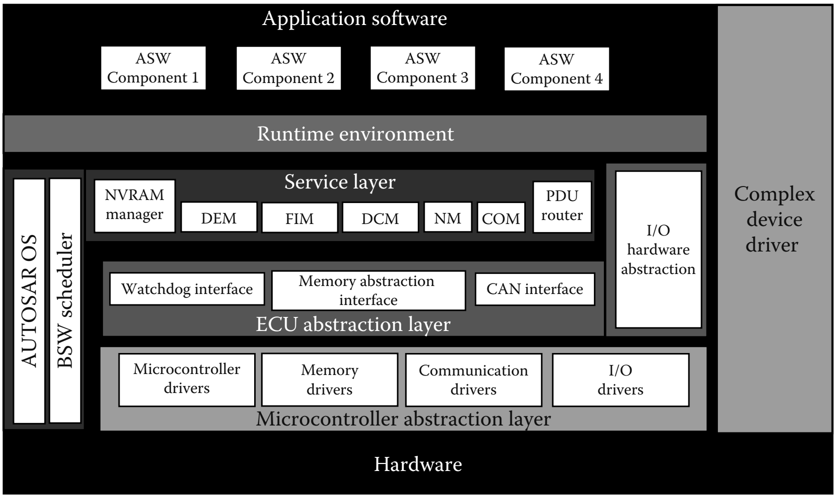


FIGURE 2.4 ECU software architecture.

the ECU abstraction layer (ECU-AL). The ECU-AL provides additional services on top of the device drivers. On top of the ECU-AL, the service layer provides additional services, such as nonvolatile random access memory (NVRAM) manager and diagnostic event manager (DEM). The AUTOSAR operating system (AUTOSAR OS) is also part of the service layer. The AUTOSAR OS must be able to access the hardware in order to manage, for example, the timer for the time-sliced scheduling. This is the reason why the service layer is allowed to access the hardware. This is shown in the shape of a flipped "L" in Figure 2.4. Besides the layered architecture, there is the so-called complex device driver, which is also allowed to directly access the hardware. The purpose of the complex device drivers is to extend the standardized part of the architecture with new device drivers, which have not yet been standardized.

The strict separation of BSW and application software (ASW) was mentioned in Section 2.1.3. This separation is supported by the RTE layer. The RTE shields the ASW from the peculiarities of the BSW and allows the ASW to access BSW services, such as the NVRAM manager, in a clearly defined way. Another important responsibility of the RTE is the provision of communication services. The RTE can be considered as a middleware that is a local realization of the concept of the VFB.

More details of the layered software architecture are described in Figure 2.4.

2.3 Main Areas of AUTOSAR Standardization: BSW and RTE

2.3.1 BSW

The AUTOSAR standard defines a fixed set of BSW modules. As described in the previous section, these modules are organized in a layered architecture. The set of

BSW modules comprises device drivers, communication and I/O drivers, AUTOSAR services like the NVRAM manager or DEM, and of course the AUTOSAR OS (Figure 2.4). In total, AUTOSAR specifies 63 BSW modules [16].

Each of these BSW modules has a clearly defined interface, which can be employed by other modules or by the RTE. Note once more that it is not possible for software components above the RTE to access any of these interfaces directly. An interface basically consists of a set of application programming interface (API) functions, a set of data types, and possible error codes returned by the API functions. Besides the interface, AUTOSAR also defines a set of configuration parameters for a BSW module. These parameters are divided into precompile, link time, and postbuild parameters, reflecting the exact point in time when the corresponding configuration takes place.

2.3.2 BSW Conformance Classes

It would be a huge effort to switch from an existing platform to AUTOSAR in one step, as this would mean implementing all 63 BSW modules, adapting the existing interfaces, and adapting the ASW to the AUTOSAR interfaces. Furthermore, during the migration period, it is expected that in the next-generation automotive systems there will be a mix of AUTOSAR and non-AUTOSAR software and ECUs.

In order to support and ease this migration, AUTOSAR defines three implementation conformance classes (ICCs) for the BSW. The basic idea is to cluster the BSW modules so that only the interfaces between these clusters have to be AUTOSAR-conform, and it is not necessary to implement each BSW module as a unit of its own. Note that the ICCs only affect the BSW and the RTE. The interfaces of the software components above the RTE are not affected. Thus, an ASW component can always be employed without changes to its interface or implementation, regardless of the ICC of the underlying RTE and BSW.

2.3.2.1 ICC1

ICC1 is the "lowest" implementation conformance class. Here, the RTE and the entire BSW are put into one cluster. Only the interface between the RTE and the ASW components and the interface to the bus have to be AUTOSAR-conform. The interface between the RTE and the BSW is not standardized in this case. Therefore, the RTE implementation is proprietary.

Nevertheless, an ICC1 implementation of BSW and RTE still has to provide the functionality and behavior as standardized by AUTOSAR, for example, the scheduling of the runnable entities (Section 2.3.3.1) of the ASW components or the communication between ASW components has to be the same as if the BSW modules were not clustered. Furthermore, ASW components expect certain functionality from the BSW, especially AUTOSAR services like the NVRAM manager or DEM. This functionality has to be provided, although not necessarily, in the form of separate BSW modules.

An ICC1 implementation would typically be the first step in migrating from an existing proprietary implementation to AUTOSAR.

2.3.2.2 ICC2

In ICC2, logically related modules are bundled into separate clusters, for example, all communication-related modules form one cluster. The RTE is one cluster on its own. The interfaces between the clusters, as well as the interfaces to the ASW components and to the bus have to be AUTOSAR-conform.

ICC2 allows for integrating BSW clusters from different vendors, for example, one could use a communication stack from vendor A and an operation system from vendor B.

2.3.2.3 ICC3

ICC3 implements the "highest" level of AUTOSAR compatibility. Here, all BSW modules as defined by AUTOSAR are present with their corresponding interfaces. There is no clustering of modules.

2.3.3 RTE

2.3.3.1 Features of RTE

As described in Section 2.2.2, there are basically two separate parts in the AUTOSAR architecture, the one above the RTE and the one below (Figure 2.5). In the part below

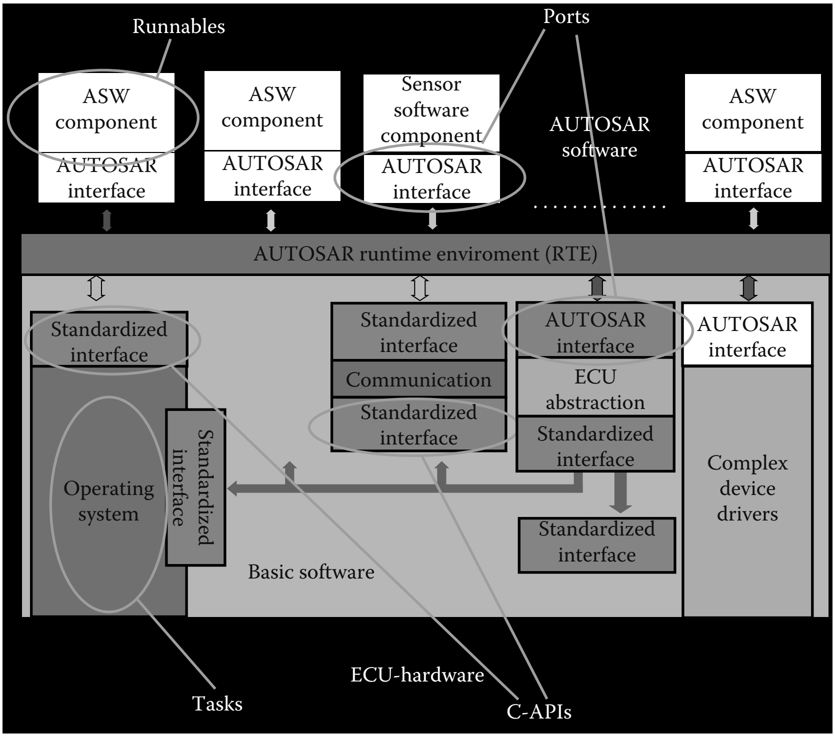


FIGURE 2.5 RTE features.

the RTE, BSW modules are free to call any API function of other modules or to use, for example, certain OS services directly. In the part above the RTE, ASW components communicate with each other via ports. There is no other way of communication (e.g., via shared global variables) allowed. An ASW component is also not allowed to use any BSW module directly. Furthermore, the dynamic behavior of an ASW component is described and implemented by means of "runnable entities." A runnable entity is a schedulable unit of an ASW component. Basically, it is a sequence of instructions that can be started by the RTE, as a result of an event initiated by the RTE. Such an RTE event is triggered, for example, when new data arrives at a port, when a timer expires, or when a server call returns. The concept of runnable entities and their activation is described in Refs. [14,17].

The task of the RTE is to glue these two parts together. The word "glue" is really important in this context. It should be made clear that the RTE is not just an abstraction layer between ASW and BSW. In a non-AUTOSAR application, the ASW typically employs OS services (like activating a task) directly or it directly sends out or receives a CAN message. This is not possible in AUTOSAR. An ASW component simply does not know the concept of an OS task or a CAN message. Also, there is no one-to-one mapping between these concepts and AUTOSAR. Therefore, it would not be sufficient to just create a wrapper around an existing proprietary application in order to make it AUTOSAR-conform. Instead, the entire internal behavior has to be adapted to the AUTOSAR paradigm.

Thus, the RTE employs the BSW in order to implement the behavior of the ASW components specified by means of ports and runnable entities. This includes two main tasks: implementing the communication and implementing the activation of runnable entities.

For the communication task, the RTE provides a set of APIs for sending or receiving data elements and for remote server calls in the case of client/server communication. Runnable entities are mapped to OS tasks (Section 2.4.5). Therefore, in order to activate a runnable entity (e.g., because data that the runnable entity was for has arrived), the RTE would typically activate the corresponding OS task that the runnable entity is mapped to. But it is also possible, in the case of a client/server operation, to call a runnable entity in the form of a direct function call.

The RTE is furthermore responsible for ensuring the consistency of data during communication, that is, to ensure that data are not changed while being received or sent.

2.3.3.2 Generation of RTE

The RTE is generated in order to ensure that it fits a given ECU and system configuration. This means that an RTE implementation always provides only the functionality that is needed for a given configuration, and nothing more. The generation process is divided into two phases (Section 2.4.5):

- Contract phase: This phase is ECU-independent. It provides the contract between a given ASW component and the RTE, that is, the API that the ASW component can be coded against. The input for this phase is the description of an ASW component with all its ports and runnable entities.

The result is an ASW component-specific header file that can be included by the corresponding source code file. In this header file, all RTE API functions that may be used by the ASW component are declared. It also declares the necessary data types and structures needed by the ASW component. The set of allowed API functions depends on the ports of the given ASW component. For example, if an ASW component has a send-port p with a data element d, the contract phase will generate the API function Rte_Send_p_d. The ASW component uses this function to send data element d via port p.

- Generation phase: In this phase the concrete code generation for a given ECU is performed. Input for this phase is the ECU configuration description, which includes especially the mapping of runnable entities to OS tasks or the communication matrix. Together with the ASW component header files created during the contract phase and all necessary BSW code, the generated code can then be compiled to an executable file for the given ECU.

Note that it is also possible to deliver an ASW component only in the form of object code, for example, in order to protect intellectual property. All necessary information is ECU-independent and already available in the contract phase. With the ASW component-specific header file it is possible to compile the source code of a given ASW component. The resulting object code together with the header file can then be delivered as a bundle to the customer.

However, the object code leaves less potential for optimizations, for example, certain functions cannot be implemented as C-macros, which would be possible if the source code of the ASW component is available.

2.4 Main Areas of AUTOSAR Standardization: Methodology and Templates

2.4.1 Objectives of the Methodology

AUTOSAR is pursuing precise technical goals to manage future software architectures. Some of these goals are

- Transferability of functions from one ECU to another ECU within the network
- Integration of functional modules from multiple suppliers
- Reuse of proven solutions (hardware and software)

In order to reach these goals, AUTOSAR has introduced a standardized architecture defined by a metamodel. This will be the basis for the interoperability of products developed using the AUTOSAR standard and a methodology approach for developing according to and complying with the AUTOSAR standard.

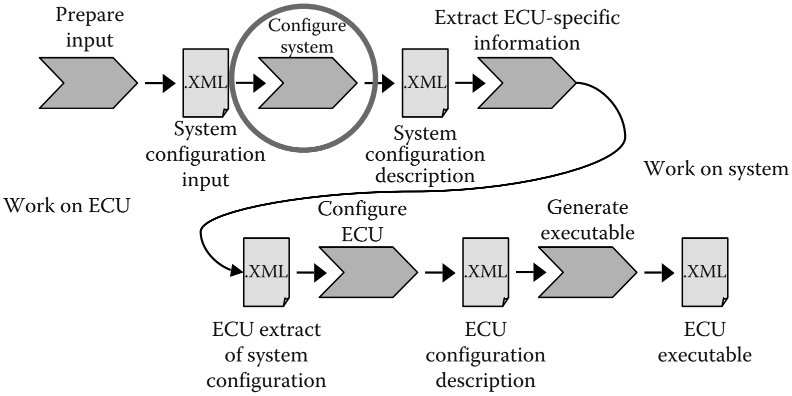

Prepare input
.XML
System configuration input
Configure system
.XML
System configuration description
Extract ECU-specific information
Work on system
Work on ECU
.XML
ECU extract of system configuration
Configure ECU
.XML
ECU configuration description
Generate executable
.XML
ECU executable

FIGURE 2.6 Overview of AUTOSAR methodology.

2.4.2 Description of the Methodology

The AUTOSAR methodology describes the dependencies of activities on work products in software process engineering metamodel (SPEM) notation. It focuses on workflow rather than specifying a full process or business interactions. It allows for consistency with a full integration into the AUTOSAR metamodel. The AUTOSAR metamodel defines how something is described, the AUTOSAR methodology defines when these descriptions are used in specific activities (Figure 2.6).

2.4.3 AUTOSAR Models, Templates, and Exchange Formats

AUTOSAR is based on models. Everything in an AUTOSAR system needs to be described in terms of standardized model elements. The models are not fixed. Information is added both in consecutive stages performed by different roles as well as iteratively by the same role.

The models are serialized to a standardized XML format for exchange and persistence using XML standard.

The source code is generated directly based on the model (ASW component API, RTE middleware, BSW configuration) (Figure 2.7). So, AUTOSAR models are not merely the documentation of the electronic/electric systems but they also drive the software development.

2.4.4 System Configuration

First, the system configuration input has to be defined (Figure 2.8). This is made by selecting ASW components and hardware, and by identifying the overall system constraints. This requires engineering decisions at the system level, which means in practice, that the appropriate templates should be filled out. The next step, the activity configure system, mainly maps the ASW components to the ECUs regarding resources

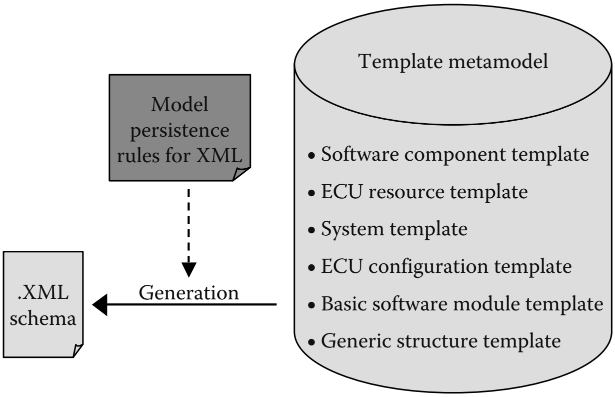

Template metamodel
Model persistence rules for XML
• Software component template
• ECU resource template
• System template
• ECU configuration template
• Basic software module template
• Generic structure template
.XML schema
Generation

FIGURE 2.7 Overview of AUTOSAR information types.

and timing requirements. This is one of the most important decisions made during the configure system activity. The output of the activity configure system is

- The system configuration description including all system information (mapping, topology, etc.)
- The allocation of each ASW component into an ECU
- The system communication matrix, which describes precisely the features of the networks/media used

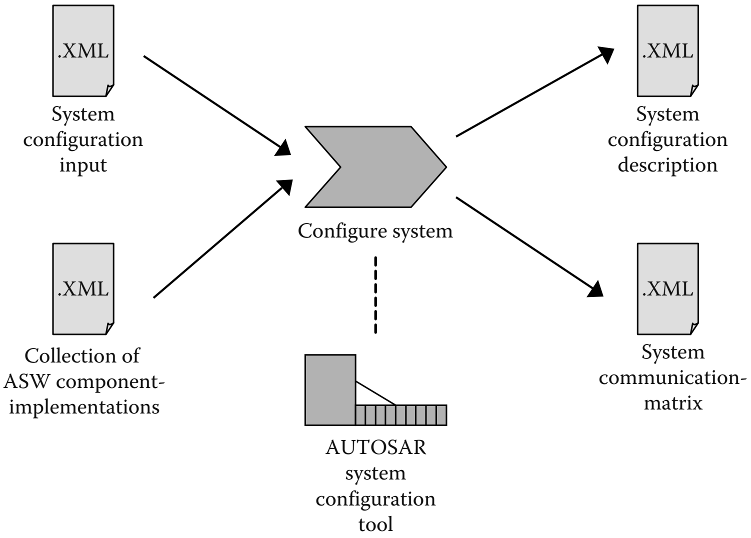

.XML
System configuration input
.XML
Collection of ASW component-implementations
Configure system
AUTOSAR system configuration tool
.XML
System configuration description
.XML
System communication-matrix

FIGURE 2.8 System configuration overview.

The system configuration generator supports all these operations. Of course, these different steps are iterative as the system design can evolve and be improved during the project development (new ASW components, new networks, new ECUs, etc.).

2.4.5 ECU Configuration

The activity "extract ECU-specific information" simply extracts the information from the system configuration description needed for a specific ECU into an ECU extract of system configuration file.

The next activity, "configure ECU", adds all necessary information for implementation like tasks, scheduling, main BSW modules list, assignments of the runnables to tasks, and configuration of the BSW modules. This activity is a nontrivial design step as it should fix all configuration parameters based on the vendor-specific and generic parameters, the BSW module description, and the collection of available ASW components implemented on the ECU. The result is included in the ECU configuration description. Due to the high complexity of this step, it has to be supported by different tool-related editors. To automatically generate parts of the configuration code for the RTE, OS, and COM, certain generators should be used.

The generate executable activity is mostly done as current executable generation with compile and link phases from designed code.

The phase "work on ECU" is also iterative depending on new ASW components that will be integrated into the ECU or new network constraints. There is a strong link between the system and ECU activities, with numerous exchanges between the different actors. But the AUTOSAR methodology does not define who is doing what and when in a software development (Figure 2.9).

2.4.6 Implementation to Existing Development Processes and Tooling

AUTOSAR defines a set of standard data types, interfaces, component types, and BSW parameters specified with models. These models will be exchanged and reused between suppliers and customers during the life cycle of the software development for implementation. An organization and a sharing of the different roles and tasks will be identified and contracted between the original equipment manufacturers (OEM) and their suppliers using interoperable tools to manipulate these models.

In order to benefit from AUTOSAR, the methodology needs to be applied to the development process. The software architecture needs to be mapped to the AUTOSAR metamodel. Former "paperwork" specifications are replaced by models,

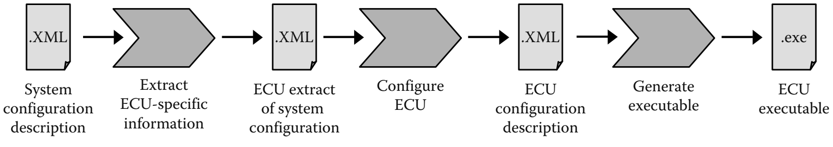

.XML
System configuration description
Extract ECU-specific information
.XML
ECU extract of system configuration
Configure ECU
.XML
ECU configuration description
Generate executable
.exe
ECU executable

FIGURE 2.9 Work on ECU.

which directly drive software development via code generation. In order to consistently work on the model, the methodology requires a dedicated AUTOSAR tooling.

The conclusion that can be drawn is that the application of AUTOSAR development requires mapping and adaptation of the existing development processes to the AUTOSAR methodology.

2.5 AUTOSAR in Practice: Conformance Testing

If an OEM buys a piece of AUTOSAR software, he or she wants to be sure that it is implemented in compliance with the specifications defined by the standard. Only if this is guaranteed can the objectives of AUTOSAR be realized, that is, the pieces of software from different suppliers run together in the system of ECUs of a specific vehicle. Furthermore, this has to be enabled in different configurations and versions before the whole system is integrated (Figure 2.10).

To enable these objectives, AUTOSAR defines conformance tests [7]. A conformance test is a test of an implementation for conformance against the requirements of a specification. It is a prerequisite for the interoperability of modules from different suppliers. In a conformance test, the tester of a system under test (SUT) is not the implementer. This prevents incorrectness and increases the quality of the test. The test results are analyzed by a conformance test agency (CTA), which finally attests the conformance of the SUT.

To ensure the intellectual property of the implementer, the tester checks the SUT as a black box. The test only needs the object code. This enables the implementer to hide details of the implementation.

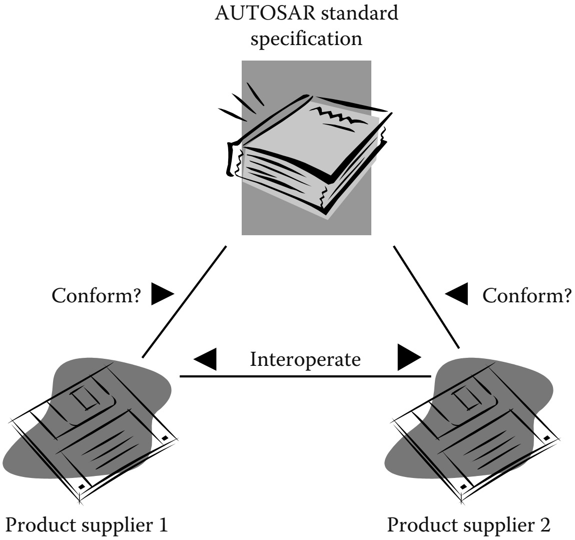


FIGURE 2.10 Conformance testing.

In a first phase, AUTOSAR concentrated on the conformance test for the BSW modules on different levels of details, the ICCs (Sections 2.3.2.1 through 2.3.2.3). Later on, the principles will be extended to include ASW components. In the following, we will restrict the considerations for BSW, that is, the SUT is always a BSW module or a cluster of BSW modules.

The objective of the tests is to check the connectability of modules and clusters in terms of basic functionality and the possibility to configure a module or cluster according to the specified set of valid parameter values. This does not include an exhaustive functional test of the correctness. Such tests, as well as integration tests, have to be operated separately. To stress this once again: the interoperability is the focus of the conformance test, not the functional correctness.

If the modules and clusters are tested to be conformant, this increases the capability to integrate them within an architecture. Interoperability among modules and clusters is supported as well as the migration from and reuse of existing solutions.

To reach these goals, the AUTOSAR standard has to deliver a consistent set of specifications for the BSW modules that are related to the SUT. This encloses the related BSW module requirements document, BSW module software specification, and conformance test specification. Based on these documents, the following steps are executed for the execution of a conformance test:

1. The AUTOSAR consortium provides the required documents (see above).
2. A CTA or a product supplier provides an executable conformance test suite.
3. Product supplier executes the tests, that is, runs the conformance test suite against the SUT.
4. Finally, a CTA approves the test results.

In all, the procedure for conformance testing enables a great amount of flexibility for the bilateral relationships between a product supplier and the party that buys the product—normally the OEM or a first-tier supplier. The supplier and the buyer can agree on who executes the test and who approves the results of such a test. If a product supplier sees an advantage for himself or herself, he or she can perform a self-declaration. In this case he or she can run the complete conformance test process in-house. On the other hand, a buyer may want to take a specific CTA.

But, this flexibility does not mean that there is a difference in the quality of the results coming from different paths. The procedure ensures a high quality of the conformance tests by the accreditation of the CTA as well as the product supplier who wants to perform the self-declaration. The demands on an AUTOSAR conformance test are compliant with ISO/IEC 17011, which describes general requirements for accreditation bodies.

2.6 AUTOSAR in Practice: Migration to AUTOSAR ECU

If a company has products in the field, the implementation of the standard in one step is not realistic, and also not advisable. A stepwise implementation linked with a guided maturity process at all development stages to ensure the quality is necessary. In fact, it is advisable to approach the AUTOSAR standard gradually to collect the first experiences at mixed platforms and to set specific, well-defined, and manageable objectives for the development. It is important to understand not only the simple use of the AUTOSAR templates and schemes, but also the basically new concepts of the AUTOSAR development process by using it to get experience.

In most cases, an AUTOSAR-conformant ECU is not developed from scratch. Existing systems are modified to reach a system that is conformant to the AUTOSAR standard.

Several kinds of mixed systems are possible:

- Some of the BSW modules may be replaced by AUTOSAR BSW modules. This does not show the applicability of the overall AUTOSAR concept. The advantage of AUTOSAR is that the whole BSW stack is standardized. With that, not only the single modules, but also the interaction between the modules is defined. The overall advantage of the specifications can be reached only if the whole AUTOSAR BSW is used.
- Clusters of BSW modules may be replaced by AUTOSAR BSW clusters. This version makes sense for getting specific advantages from the defined clusters, for example, COM stack. Since it is a usual business model these days for several vendors to sell specific clusters, such a variant fits well into the business models that existed before AUTOSAR. As one may say, it is better than having nothing.
- An RTE implementation may be added above the original, non-AUTOSAR-compliant BSW and used for some applications, but not all. This version is a possible migration path for the applications used in experimental systems. It is a first step toward reaching the ICC1 level. It is not recommended for serial production as it needs additional memory and is without any functional advantages.
- A system methodology may be used for specifying the system, but may not be used to design an AUTOSAR architecture and implementation.

  Not only does one need migration scenarios for implementing the processes, but also for adapting them into the system. As their change is often related to organizational restructuring, the change of a process takes longer than that for the architecture and may be established earlier.

In the remaining part of this section, we want to describe the necessary steps in the migration from an ECU to an AUTOSAR ECU. We present these steps in the form of a use case with a title, a pre-, a postcondition, and a step-by-step description.

Title of Use Case:

Change an existing non-AUTOSAR ECU such that it encloses AUTOSAR BSW, RTE, and ASW components.

Precondition:

An ECU with non-AUTOSAR software is available.

Postcondition:
An ECU with AUTOSAR software is available.

Description:

1. Decide on an ICC1, ICC2, or ICC3 ECU architecture. The phrase "not from scratch" encloses that usually well-trained software development processes exist. It is necessary to analyze the existing software architecture and building process. Examine how the existing BSW may interact with the RTE. Decide on which way to go. Is the existing BSW able to fulfill ICC3 with slight changes? Or is it more appropriate to develop an ICC1 system? Such a decision cannot be taken by a supplier for its own product. If the OEM intends to run ASW components from other suppliers on this ECU, the requirements coming from these additional ASW components have also to be taken into account.
2. Describe the ECU type, the connectivity, and resources using the ECU-resource template. This step can be done directly by the semiconductor vendor. The templates may be delivered together with the hardware as part of the documentation.
3. Develop architecture for the application layer. Analyze the software applications that already run on the ECU. Logically divide the applications into components. Split these components into hardware-independent and hardware-dependent components. The first group will run as ASW components on top of an RTE with the underlying BSW. The second group may be implemented as a complex device driver.

   Transform the hardware-independent components into ASW components. Extracting a single component from the legacy application means, all internal and external communication needs to be analyzed. The data types and interfaces used by the component are modeled according to this analysis. The actual component—this may be an atomic-component, a sensor-component, or an actor-component—is modeled by using these interfaces.

   Apart from its communication, the internal behavior of the component also needs to be analyzed. Cyclically based and event-based functions in the legacy code are modeled as runnable entities.
4. Go through the system configuration step. The integrator designs mapping constraints using the system constraint part of the system template. All components planned to run on the same ECU are mapped to its hardware in the system-generation step.

   The outcome of the generation step is extracted with the help of an ECU extractor. The information necessary to implement, configure, and test the single ECU is separated. Now, one can run the AUTOSAR RTE generator contract phase. This generates the set of header files needed by the application programmer to implement the component. The components are linked together with the help of ports and connectors.
5. Integrate ECU. Configure the BSW, such as thread usage and thread priorities, bus-system communication parameters (e.g., CAN frame

priorities), etc. RTE generation phase using the RTE generator. If all the necessary software for the ECU has been collected, including the BSW code, the RTE code, and the ASW components code, one can build the ECU software image. Download the image to the target and test it.

The concrete steps are not defined by AUTOSAR. A lot of developers were disappointed by this fact because they expected to get more concrete process recommendations from the standard. But it is not in responsibility of the standard to publish a suitable process. This is an arena for competition. Therefore, each company is responsible for the process. Also, the expectation to get even guidelines has to be denied. This would be interfering with the standard itself. So, the borders of AUTOSAR are very clear.

2.7 AUTOSAR in Practice: Application of OEM–Supplier Collaboration

In this section, we want to consider an example for how an OEM and a supplier may collaborate during the development of a network of ECUs. It is assumed that the supplier develops several ECUs that run within the network of ECUs in a vehicle. The OEM is the integrator for the whole system. Again, the steps are presented in the form of a use case description (Figure 2.11).

Title of Use Case:

A subsystem, consisting of several ECUs (integrated hardware and software), is sold by one supplier to an OEM. The OEM integrates the subsystem into its vehicle network of ECUs. The remaining ECUs may come from different suppliers.

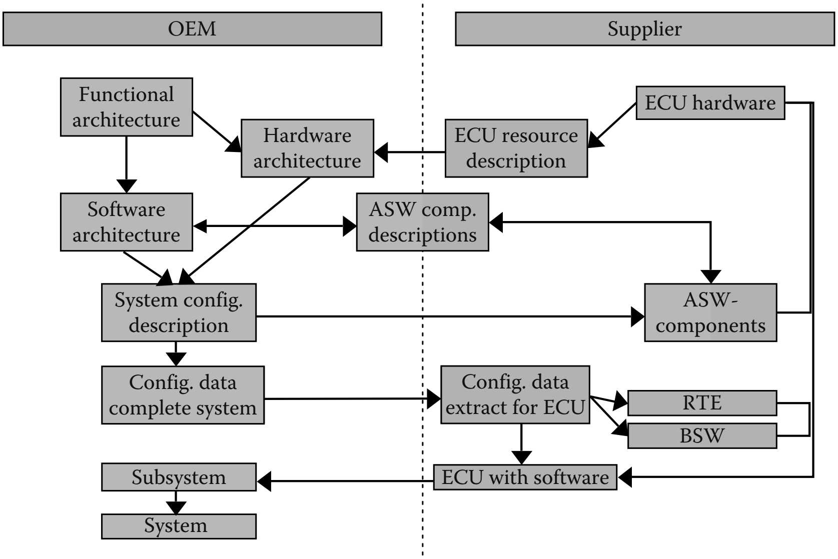


FIGURE 2.11 Example for an OEM–supplier collaboration.

Precondition:

A subsystem, consisting of several ECUs equipped with AUTOSAR software, is developed by the considered supplier.

Postcondition:

A subsystem of ECUs is adapted according to the customer's specific wishes and is sold to the customer.

Description:

1. The supplier delivers the input information for the system configuration step. The supplier delivers the ECU resource description for the hardware used in the subsystem to the OEM. The OEM adds this description to the ones used in the whole vehicle such that he or she gets a complete set of ECU resource descriptions for all hardware used in the dedicated vehicle.

   The supplier delivers the ASW component descriptions for the ASW components of the subsystem to the OEM. The OEM adds these descriptions to his or her software architecture for the whole vehicle.

   The supplier delivers the system constraints for the ASW components used in the subsystem to the OEM. The OEM integrates these system constraints into the system architecture.
2. OEM runs the system configuration. The OEM integrates all three templates—the ECU resource description, the system template, and the ASW component template. Now, the OEM runs through the system configuration step. This step is an iterative one. A first mapping of ASW components to the ECU hardware may lead to the need to adapt the system architecture as well as the software architecture. This leads to changes on the side of the OEM as well as on the supplier's side. So, a strong relationship between OEM and the involved suppliers is necessary.

   The OEM extracts the configuration descriptions for the dedicated subsystem of ECUs that will be implemented by the considered supplier. The OEM delivers this configuration data to the supplier.
3. Supplier implements subsystem. The supplier adapts its subsystem, generates the RTE, and configures the BSW for each ECU. After the integration and testing on the ECU as well as on the subsystem level, the supplier delivers the integrated subsystem to the OEM.
4. OEM integrates system. The OEM integrates the subsystem into the whole electronic/electric system, including the necessary integration and testing steps.

The use case shows the intensive interaction between OEM and suppliers. This is particularly necessary when considering early integration on the vehicle function bus level. Such a new concept breaks traditional OEM–supplier relationships. As mentioned for the development processes, the introduction of new OEM–supplier collaboration processes needs time and is difficult to establish. But the success of the whole standard depends on the success of the processes demanded by the standard.

2.8 AUTOSAR in Practice: Demonstration of AUTOSAR-Compliant ECUs

To put all the previously described concepts into practice, one needs experience. To acquire and demonstrate this experience, a learning project in the form of a demonstrator buildup is a suitable method. This section gives a brief overview of a demonstrator that shows the concepts and application of the specifications. The development itself helps to teach how the methodology can be applied. The demonstrator was developed by Siemens in the business field of Siemens VDO automotive. The demonstrator is a set of ECUs from the different subdomains in the vehicle that were considered in phase I of AUTOSAR, that is, body, chassis, and power-train.

2.8.1 Description of the Demonstrator

The main functionality that is shown in the demonstrator is cruise control. This is a function that needs data from several sensors coming from different domains, has a central (domain independent) responsibility to process the data, and uses several output devices. With that, this function has a lot of aspects that are suitable for showing several of the main objectives of AUTOSAR. But to show the interaction with other functionalities, additional applications are realized. These include air conditioning, wiper washer, and central door locking. In this section, though, we will concentrate on cruise control.

The driver can activate the cruise control function by pressing a button to set the speed. The cruise control starts to maintain the speed based on the actual vehicle speed. To inform the driver about the actual status, a cruise control light symbol is activated within the instrument cluster. The driver can revert to manual control by pressing a cancel button or the brake pedal. Once having activated the cruise control functionality, the speed is memorized. The last active value can be resumed by pressing a resume button. At any time, the driver can override the actual set speed by pushing the acceleration pedal. If the driver releases the pedal, the previously set speed is resumed and maintained again.

For the presentation of the cross-domain concepts of AUTOSAR, this functionality is in particular suitable because of

- Several sensors: acceleration pedal, brake, buttons
- One domain independent algorithm to process the data
- Several actuators: engine control, instrument cluster

The demonstrator consists of a set of four ECUs with reduced functionality and an additional PC. They are connected via a high-speed CAN. Three of them are based on NEC V850 hardware and one on a TriCore microcontroller (μC). The PC is used for the central control of the whole functionality, bus-traffic-simulation, and human–machine interface (HMI) purposes. On the V850 ECUs, an ICC3 AUTOSAR BSW stack is implemented. On the TriCore μC, an ICC2 implementation of the AUTOSAR BSW stack is realized. Each ECU takes over a specific role in the network. One takes over the responsibility of a real-time server, one of an engine control unit, one of a

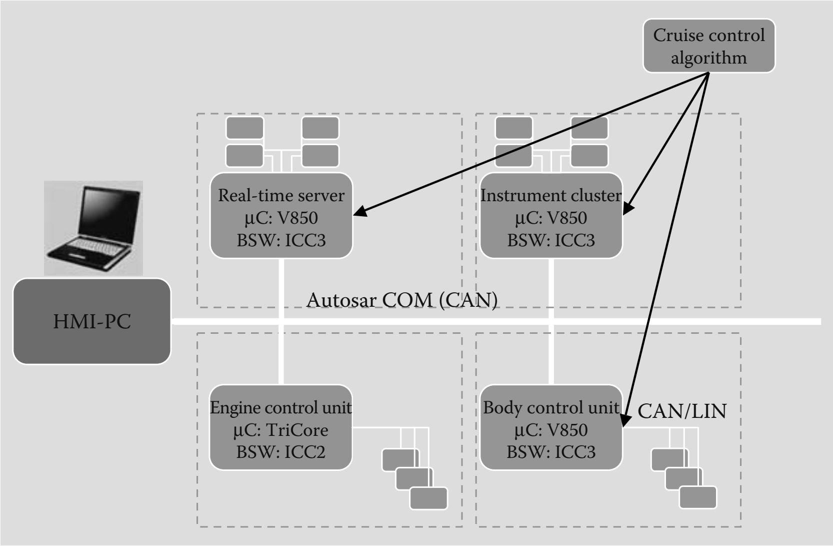


FIGURE 2.12 Cruise control.

body control unit, and one of an instrument cluster control unit. Sensors and actuators are connected via CAN to one of the ECUs (Figure 2.12).

The whole system is placed in two transportable suitcases, such that it can be used for demonstrations at the customer's site. Each of the involved subdomains can now illustrate how the specific ECU works in the cross-domain network of AUTOSAR ECUs.

2.8.2 Concepts Shown by the Demonstrator

The result shows the practice of the AUTOSAR development cycle and the ability of the integration of a cross-domain system functionality using AUTOSAR. Expertise in AUTOSAR development is gained by applying all steps of the AUTOSAR methodology, beginning with the use of the different templates and the system configuration. ASW components are described by means of the ASW component template. With that, the usability of the ASW component template is shown, and the ASW components are independent from concrete hardware.

Here the term "usability" has to be understood, and not just in the sense that one can describe an ASW component with such a template. In addition, it also shows that an existing control algorithm and actuator control can be applied to the AUTOSAR concept to describe a system. The results of this step are the AUTOSAR-compliant ASW component descriptions and the constraint description for the cruise control algorithm, the engine software, and the remaining necessary ASW components.

The system is specified and simulated on the VFB level. One of the objectives of AUTOSAR is to enable the simulation of ASW component behavior on a level independent of the dedicated hardware. This is achieved by a simulation with a prototypical simulation tool. Such a simulation enables an early integration and validation to be shown on a model level, for example, all engine and vehicle sensor- and actuator-I/Os needed to run the engine control unit are simulated. These components were simulated in a hardware-in-the-loop (HIL) simulation as well as in the VFB simulation, which simulated the ASW components and the RTE. This enables the two simulation techniques to be compared directly.

After this virtual integration, the ASW components are mapped to dedicated, real hardware and integrated together with the AUTOSAR BSW and certain additional complex device drivers. To show the independence of the cruise control ASW component implementation, this ASW component can be mapped to all three of the ECUs that the ICC3 BSW system runs on.

The implementation of the BSW is based on the release 2.0 (R2.0) of AUTOSAR. The software that runs on the engine control unit has specific requirements, which were not included in R2.0 of the AUTOSAR specification. Therefore, the BSW on this ECU consists of AUTOSAR as well as non-AUTOSAR BSW modules. In all, this construction also shows the applicability of the AUTOSAR concepts for the powertrain domain.

The integration incorporates the configuration of the BSW and the ASW components, as well as the generation of an RTE for the concrete target hardware by an RTE generator tool.

To summarize the additional concepts shown by the implementation: it shows the usage of the BSW and RTE implementations

- On different hardware platforms
- For mode management, especially on low power mode
- For combining AUTOSAR BSW with complex device drivers

This implementation also enables the connection of sensors and actors between ECUs to be shifted. This can be realized as a pluggable or as a deployment scenario, that is, a shift between LIN and direct connections.

2.9 Business Aspects

From a business point of view, AUTOSAR changes the whole market with ECUs and the software for them. But AUTOSAR is not the cause of these changes. AUTOSAR only applies and speeds up the trends in modern software development. The main aspect in these trends is the separation of ASW from the hardware by introducing layered architectures. With this, AUTOSAR enables a finer granular business. The customer can choose the parts of the system independently and build up his system more flexibly. Let us consider each small part individually.

BSW: As the functionality of the BSW is standardized on a detailed level, the BSW will be a commodity product. The differentiation between the vendors is no longer given by the specified functionality. The differentiation is restricted to an optimization of the implementation with respect to performance and memory usages.

RTE: The RTE is not a manually developed software, it is generated. The RTE generator will probably only be a piece of a tool integrated into a bigger system-development tool. Or it will be given in addition to the BSW if the BSW is delivered as one block and not in individual modules. So, this part will probably not form a separate business, but is a necessary part to run the business with BSW and the related tools.

ASW: As the ASW is independent from the underlying hardware, more flexibility is given for business with this part of the software. The main differentiation in the software market is assigned to the ASW. It enables more flexibility for the customer to choose software that best fits his or her needs. On the other hand, it gives the suppliers more flexibility in differentiating from their competitors.

In the past, due to the coupling of hardware and software, this flexibility was limited because of technical restrictions. With AUTOSAR, the flexibility for new functionality will be driven more by the software.

Modeling tools: The specification and implementation of tools is not part of AUTOSAR. AUTOSAR standardizes the exchange formats on a model level. This has a major influence on the tool market. It enables the use of different tools for both the customer and the product supplier. This ensures that each party has more flexibility in their choice of tools. It is no longer necessary for a product supplier to work with specific tools for each customer. The coupling between tools and BSW, which was a usual market model in the past, is clearly restricted with AUTOSAR.

System integration: AUTOSAR eases the integration of all components, modules, and subsystems into an overall vehicle system. The differentiation in the car market will be driven by building new functions through the integration and cooperation of vehicle-wide distributed functions.

2.10 Outlook

The core partners agreed to continue with the project partnership from 2007 to 2009. This second phase will maintain and extend the deliverables of phase I, but will also bring new features and specifications. The main objectives of phase II are

- Improve and extend the AUTOSAR software architecture and communication mechanisms (error handling, extension of the VFB capabilities, debugging facilities on RTE level, support for multimedia requirements, concept for vehicle mode management, etc.).
- Implement the safety concepts specified in phase I.
- Complete and finalize the work on AUTOSAR BSW modules (communication stacks, diagnostic, time supervision).
- Continue to standardize the interface on the application level for body and comfort, power train, chassis, pedestrian and occupant safety systems, multimedia, telematics, and HMI.
- Maintain and support the AUTOSAR specifications and enable the exploitation of AUTOSAR within the consortium.

All new concepts will be, as in phase I, proved by validators and demonstrators.

The exploitation of the standard starts in parallel to the second phase. One can see in the press, at conferences, and fairs, that the number of AUTOSAR-related products is increasing very quickly. The experiments done during these early developments will bring further change requests to the standard. The members of the consortium are working on important exploitation that will bring AUTOSAR to industrial maturity.

References

1. ITEA, *Technology Roadmap on Software Intensive Systems*, ITEA Office Association, Eindhoven, the Netherlands, March 2001.
2. K. Jost, Electronics demand to grow nearly 7% annually, *Automotive Engineering International*, September 2001.
3. ATKerney, Software-betriebene Fahrzeugsysteme bestimmen die Zukunft des Automobils Study, 2001.
4. McKinsey & Company, *Automotive Software: A Battle for Value Study*, 2002.
5. D. Sallee and R. Bannatyne, Trends in advanced chassis control, *Automotive Engineering International*, September 2001.
6. K. Jost, From fly-by-wire to drive-by-wire, *Automotive Engineering International*, September 2001.
7. H. Fennel, AUTOSAR—a standardized automotive software architecture, OOP Conference Presentation, January 2007.
8. AUTOSAR GbR; Achievements and exploitation of the AUTOSAR development partnership, Convergence Conference Paper, Detroit, October 2006.
9. AUTOSAR GbR, AUTOSAR—the standard, its exploitation and further development, AAET 2007—Automatisierungs-, Assistenzsysteme und eingebettete Systeme für Transportmittel, Conference paper, Braunschweig, February 2007.
10. S. Voget, AUTOSAR standard, Embedded World Conference Presentation, Nürnberg, February 2007.
11. AUTOSAR GbR, *Main Requirements, V2.0.1.* Available at: www.autosar.org, December 2006.
12. AUTOSAR GbR, *AUTOSAR Meta Model, V1.0.1.* Available at: www.autosar.org, AUTOSAR_MetaModel.zip, December 2006.
13. OMG, *Meta Object Facility (MOF) 2.0 Core Specification*. Available at: www.omg.org.
14. AUTOSAR GbR, *Software Component Template, V2.0.1.* Available at: www.autosar.org, December 2006.
15. OMG, *Common Object Request Broker Architecture (CORBA/IIOP), V3.0.3.* Available at: www.omg.org.
16. AUTOSAR GbR, *List of Basic Software Modules, V1.0.0.* Available at: www.autosar.org, December 2006.
17. AUTOSAR GbR, *Specification of RTE Software, V1.0.1.* Available at: www.autosar.org, December 2006.

3

Intelligent Vehicle Technologies

Michel Parent
National Institute for Research in Computer Science and Control

Patrice Bodu
National Institute for Research in Computer Science and Control

3.1 Introduction: Road Transport and Its Evolution

3.1.1 Such a Wonderful Product

Throughout the twentieth century, the automobile and its infrastructure were developed in such a way as to become the dominant mode of transport for passengers and goods in most industrialized countries. In these countries, a level of about one vehicle per person (around 800 vehicles per 1000 inhabitants) has been reached while in countries like China and India, the current level is about 25 vehicles per 1000 inhabitants but growing all the time [1].

We can therefore say that the automobile has probably been the most successful and influential product of the twentieth century. It has created an enormous industry worldwide; it has changed the lives of millions of people and also changed the way cities are organized.

However, this extreme growth has brought about several problems that we now have to face. The problems mostly concern the safety, the congestion of infrastructures, and the energy needed for all these vehicles. As we will see in this chapter, new electronic technologies are now bringing some solutions to these problems.

3.1.2 Safety Problems

The number of deaths on the roads has reached the astonishing level of more than 1 million per year worldwide. This is a problem of much greater magnitude than any past war, and governments in most industrialized countries have addressed this as a major challenge. Many countries have set a target of reducing the death rate by 50% over the next 5 years and, in some places such as Sweden, a target of "zero death" has been set [2].

Vehicles and infrastructures have already been greatly improved but now the proportion of accidents due to driver error has increased. Consequently, new control techniques are being developed in order to take the control away from the driver in order to minimize his or her errors [3].

3.1.3 Congestion Problem

Congestion of infrastructures is a major problem in most conurbations and on major corridors. It is costing several percentage points of gross national product (GNP) in many countries in lost time and energy. Furthermore, congestion leads to an increase in pollution and greenhouse gases (GHGs).

Improvements in vehicle quality have helped to increase capacity, for example, through reduced safe-stopping distances and improved acceleration, but the automobile is still very inefficient in terms of space usage, in particular in its private form (where it stands still for most of the time).

In order to meet a continuously growing demand for transport, the solutions for industrialized countries now lie in better management of the resources (infrastructures and vehicles), in better use of intermodality (the optimal use of different forms of transport since mass transport is unavoidable in large conurbations) but also in new technologies for vehicle control.

Indeed, the basic control techniques for vehicles have not changed much in the last 100 years, with the driver having the total responsibility of his vehicle through mechanical impediments (steering wheel and pedals). These primitive controls lead to inefficiencies and accidents. New control techniques such as adaptive cruise control (ACC), if properly designed, could definitely improve the throughput of infrastructures by reducing the time gap between vehicles and also by introducing a smoother flow with fewer "shock waves," which lead to traffic jams [4].

Another type of control needs to be implemented to regulate the demand and avoid congestion (which leads to reduced capacity). The general tendency here is to introduce some form of road pricing (or congestion charging) in order to reduce the demand locally when the traffic approaches congestion levels [5].

3.1.4 Energy and Emissions

Individual vehicles are, in fact, quite efficient in terms of energy used per passenger/kilometer and can be compared to other modes on the average [6]. Of course, mass transport can be very efficient when fully loaded but since they sometimes run almost empty (and completely empty of passengers for return trips to the depot), their efficiency is often not much better than an individual vehicle with its average occupation of about 1.2 passengers.

The main problem of individual vehicles is the type of fuel they use. Most of the vehicles on the world market use fossil fuels, which lead to various types of emissions. Although great progress has been made by the automobile manufacturers to reduce the local pollutants, there are still some problems left with NOx and particulates. However, the biggest problem concerns the emissions of CO_2 and their potential effect on the planet (greenhouse effect). This is becoming a major issue worldwide and with the increase of road transport (in particular, for goods but also for passengers in developing countries), the targets for reducing GHGs are almost impossible to meet in the short term.

The medium-term solutions seem to lie in the same direction as those for the use of infrastructures with a better modal split between mass transport and individual transport, and better control of vehicle use through road pricing schemes. In the longer term, new fuels such as biogas or hydrogen (produced through carbon neutral schemes) may provide a comprehensive solution to GHG emissions.

3.1.5 Conclusion and Presentation of the Chapter

In conclusion, we see that the automobile as it evolved through the twentieth century is about to change radically in order to meet several challenges:

- Better safety
- Better use of energy
- Better efficiency in terms of usage of space

Key technologies to meet these challenges will be presented in this chapter and will include new developments on sensors, actuators, and control technologies. However, a key factor for the introduction of these technologies, which tend to take the control of the vehicle away from the driver, will be their reliability, their acceptance by the users, and the regulations that will allow or impose them on the road.

3.2 New Technologies

3.2.1 Sensor Technologies

Sensors are the essential elements in any control system and this is particularly true for road vehicles when we want to introduce some form of assistance to the driver. Here, we will give a brief overview of the existing sensors now in use in the industry.

As seen in Figure 3.1, sensors operate at different ranges for various applications. Ultrasound sensors provide the opportunity to assist low-speed maneuvers such as parking, while cameras and lasers provide sufficient range for city driving assistance, and radars are usually used for detecting vehicles ahead while driving at high speed.

3.2.1.1 Ultrasound Sensors

These simple and cheap active sensors emit a cone-shaped ultrasonic wave through the electric actuation of an electrostatic or piezoelectric transducer and receive the wave's echo through symmetrical transduction. Measuring the duration between the emission and reception time gives an estimation of the distance of the nearest obstacle, with a maximum detection range of a few meters [7]. Ultrasound sensors can also be used for angular position estimation, as seen in Ref. [8].

Ultrasonic sensors are currently in general use in the automotive industry for a few applications, the most common being the back maneuvers and parking assist systems, as well as intrusion detection systems. In Europe, every major car constructor proposes these options in mid- and top-of-the-range vehicles. However, these sensors just give an audio feedback to the driver and there is no application yet where they are involved in the control of the vehicle. This may change in the future with parking assistance where the sensors may be used to detect the exact space available for a maneuver and in the execution of the maneuver.

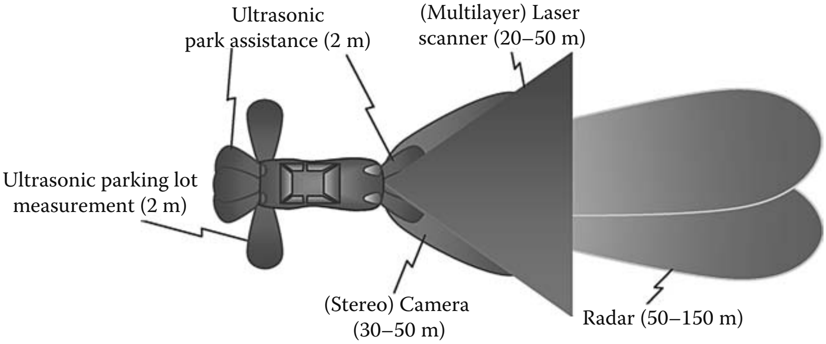


FIGURE 3.1 Sensor implantation and range overview.

3.2.1.2 Inertial Sensors-Accelerometer-Gyrometers

Full inertial measurement unit is comprised of six sensors allowing measurement over the six degrees of freedom (DoF) of a vehicle, namely three orientations (roll, pitch, and yaw) and three accelerations.

However, since the vehicle operates on a road (as shown in Figure 3.2: the $(\mathrm{O}x, \mathrm{O}y)$ plan), the DoF estimation needed for localization issues can be reduced to angular rotation around the vertical axis (yaw) and longitudinal acceleration measurements, which are sufficient to reconstitute an approximation of the vehicle's trajectory. Measuring the wheel rotation and steering angle is a cheap alternative if the wheels are not sliding.

3.2.1.3 Light Detection and Ranging or Laser Detection and Ranging

Light detection and ranging (LIDAR) or laser detection and ranging (LADAR), which is often the term used in the military field, are active sensors consisting of a light source, a photon detection system, a timing circuit, and optics for both the source and the receiver. LIDAR sends an amplitude-modulated continuous signal and determines the phase shift of its echo [9].

With a fixed sinusoidal frequency f and if there is an object at distance d, a phase shift of $\Delta\phi = 2\pi f(2d/c)$ will be observed between the transmitted signal and the received signal, with c representing light speed. An estimation of the object's distance is given by $d = \Delta\phi\, c/4\pi f$.

The maximum distance that can be estimated through phase-shift measurement is given by computing $d_{\max} = (2\pi c/4\pi f) = (c/2f) = (\lambda/2)$, where λ designates the signal's wavelength. Beyond that distance, the number of phase revolutions becomes indeterminate (Table 3.1)

Since infrared signals have a micrometric wavelength, amplitude modulation (Figure 3.3) is used to bypass the limitations of phase-shift measurement by using the

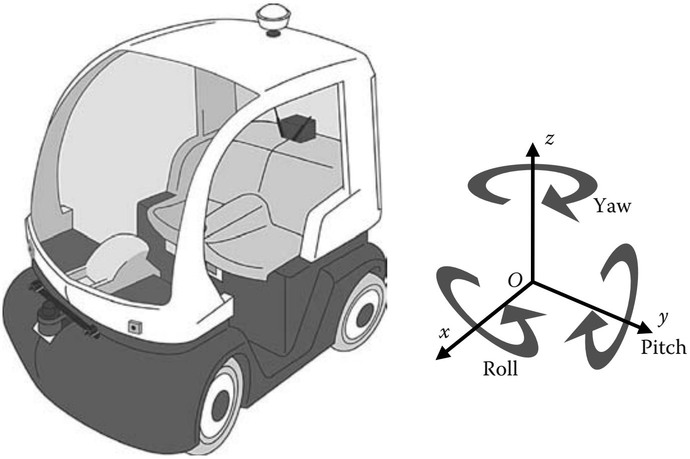


FIGURE 3.2 DoF of a ground vehicle.

TABLE 3.1 Maximum Distance Using Phase-Shift Estimation

Signal Frequency	Maximum Range
1 kHz	150 km
1 MHz	150 m
1 GHz	15 cm

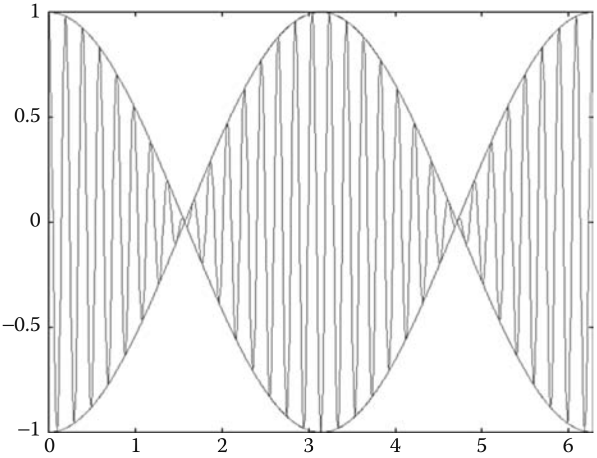


FIGURE 3.3 Amplitude modulation.

phase of the envelope while keeping the advantages of the propagation of the infrared carrier.

Another LADAR technique is to send pulses, receive reflected waves, and measure the duration of the back-and-forth trip. Pulses are modulated in amplitude by pseudorandom coding, to facilitate the association of outgoing and incoming signals.

Using a rotating head (on one or two axes) or using mirrors, several scans are performed so that the range of obstacles is determined in numerous directions. A trade-off between resolution, aperture, and sweep rate is chosen so that satisfactory performance is obtained for the application. In good weather conditions, allowed laser sources can detect obstacles up to 200 m with a few centimeters' precision.

The major problem with such scanning devices comes from the cost and reliability (over the lifetime of a vehicle) of the mechanical components. Micromechanical technologies may point the way onward. With reduced size comes reduced inertia, which in turn permits higher performance [10]. Micromirror arrays could prove to be a very useful technology for controlling the resolution of LIDAR sensors. Micromirrors can also act as a distributed scanner, generating a large number of microbeams that can scan the workspace from different angles and positions.

3.2.1.4 Radio Detection and Ranging

Radio detection and ranging (RADAR) is an active sensor that sends a high frequency electromagnetic wave and immediately receives its echo in turn, which is processed so that the range, azimuth, and velocity of the obstacle are determined [11].

In pulsed mode, the relative speed (V_r) of the target is measured using the Doppler effect, where frequency shift Δf equals $2V_r f/c$. Range measurement is obtained through an estimation of the reflected signal strength, with a proportional factor of $1/d^4$.

Like LIDAR, continuous-wave radars are also employed, but this time using frequency modulation, so that the ambiguity introduced by the Doppler shift is counterbalanced by adequate frequency-domain processing. At high frequencies (ca. 77 GHz), frequency-modulated continuous waves (FMCW) technology tends to be more economical, since pulsed transmission control requires expensive components. FMCW also offers very short-range capability since echo is captured continuously, whereas pulsed chirp applications need recovery time after an impulsion [12].

Automotive radars can range up to 150 m with a 12° search area. They can measure up to a relative velocity of 60 m/s (215 km/h) with 1‰ precision.

3.2.1.5 Vision Sensor

Vision sensors offer a 2D array of up to a million pixels with a wide field of vision, the angular field of vision depending upon the optics. Complementary metal–oxide semiconductor (CMOS) imagers tend to present more advantages than charge-coupled device (CCD), since they have a wider (nonlinear) luminance range, lower power consumption and cost, and individual pixel-processing facilities [13]. System on chip (SoC) technologies help to design integrated devices that quickly output preprocessed primitives, which are then handled by higher level applications [14].

Stereo-vision systems analyze two snapshots taken from slightly shifted points of view. Appropriate algorithms match pixels in both snapshots and calculate the "disparity map," which tracks pixel shift between the two images (this shift is horizontal when the cameras have parallel lines of sight). If the stereo cameras have been calibrated, it is possible to reconstruct the spatial distribution of objects from the disparity map.

The gauge between the cameras' optical centers—the baseline—determines the effectiveness of the distance estimation. A close implantation of the cameras will provide precise short-range estimation but small maximum range, while a large line base will provide better maximum range, at the expense of short-range precision.

Automotive applications using cameras started to appear on the market at the end of the 1990s. One of the first commercial applications was lane departure warning in Mercedes trucks in 2002, and in C4s and C5s from Citroën in 2005, where the warning is performed by a "haptic device" (vibrating seat). Night-vision modules hit the market in 2006, with the Mercedes S-class. However, most of the applications, here again, focus on information brought to the user and no real control.

Stereo-vision obstacle detection systems are being studied by original equipment manufacturer (OEM) suppliers, constructors, and researchers, as seen in some of PReVENT's subprojects, such as APALACI, that aims at detecting vulnerable road users and pedestrian safety and risk mitigation [3].

3.2.1.6 Global Navigation Satellite System

The global positioning system (GPS) is a system launched in 1980 by the Department of Defence of the United States. It provides information on time, position, and velocity at any location on the planet. A constellation of 24 satellites has been evenly spaced at 20,200 km altitude, in circular 12 h orbits, and inclined 55° to the equatorial plane, to provide at a reasonable cost an Earth-wide coverage. Satellites use atomic clocks to keep consistent timelines. They are able to transmit two microwave carriers, with 1227.60 and 1575.42 MHz frequencies, respectively. The Russian Federation has launched a similar position system in 1982 called Glonass, while Europe is testing the Galileo system since the end of 2006, and planning to release it in 2010. Galileo is a joint initiative of the European Commission and the European Space Agency (ESA), aiming at civilian applications, and is financed by these institutions and two private consortiums: Eurely (EADS/Thales/Inmarsat) and iNavSat (Alcatel/Finmeccanica/AENA/Hispasat). Galileo's 30 satellites will diffuse a public localization service, up to a precision of 5 m, a commercial localization service up to a precision of 1 m, and some services for critical civilian applications.

The basic principle of global navigation satellite system (GNSS) is triangulation. If the receptor can estimate the distances between itself and several satellites whose location is well known, it must be somewhere inside the volume, defined as the intersection of the spheres centered on each satellite with a radius equal to the corresponding estimated distance. The latter is given through estimation of traveling time between an emitting satellite and the receptor. The emitted message contains a timestamp that is compared to an absolute reception time, the receptor being synchronized with the constellation. This timing has to be very precise, since, at light speed, a 1 μs error leads to 300 m shift.

In Figure 3.4, the localization of the GPS receptor is computed by using a priori information: the positions p_1, p_2, p_3 that are transmitted by the satellites, and the estimated pseudoranges r_1, r_2, r_3, obtained from the signals' traveling time. The receptor is located at the intersection of the spheres, centered respectively on p_1, p_2, and p_3, with radiuses respectively equal to r_1, r_2, and r_3.

In order to get the precise position of the spheres' center position, the satellites transmit their real-time orbits so that the deviation from nominal orbits is not passed on to the receptor's position estimation. Moreover, in order to get the precise estimation of the spheres' radius, receptor and satellite clocks must be synchronized through complex distributed algorithms that reduce the receptor's clock drift.

Local base stations can provide additional information to correct errors with various fine-tuning factors, such as the ionospheric (50–500 km altitude) refraction, which slightly deforms the light's trajectory and introduces shifts between the electromagnetic wave's actual path and the simplified straightforward path. Ionospheric

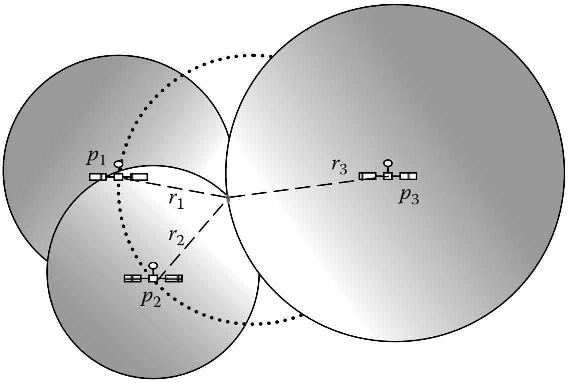


FIGURE 3.4 GPS triangulation.

corrections have to be continuously broadcasted, since the geometry of the problem is in permanent, though slow, evolution. GPS localization systems receiving such correction from a station are known as differential GPS (DGPS).

Another common source of errors is occultation/reflection of electromagnetic waves, since, if direct rays are stopped by an obstacle while reflected ones still reach the receptor, the length of the indirect path will be used for the computation, which can lead to hops of the estimated position. Low-pass filters are usually used to minimize any sudden evolution of position estimation.

Common automotive GNSS sensors provide 10 m precision position measurements. Using local corrections transmitted to the receiver (DGPS), a precision of 1 m is easily obtained, while high-end receptors such as real-time kinematic (RTK) GPS provide centimetric estimations.

3.2.2 Sensor Fusion

3.2.2.1 Introduction

Sensor fusion is the process that uses multiple sensors to provide an estimation of the vehicle's state and its surroundings. The main difficulty in data fusion is aggregating data that often have asynchronous timelines, and provide partial and noisy sensor data. If the data fusion algorithm was to synchronize all data before iterating one cycle, it could introduce high latency, at the expense of system controllability or reaction delay.

3.2.2.2 Sensor Fusion for Improved Localization

The localization problem is a good example of heterogeneous sensor data fusion. The sensors used to localize the system's position can be absolute or relative. Absolute position sensors such as GNSS or artificial landmarks provide bounded incertitude, but usually have insufficient precision or refresh rates for automotive control applications. Relative position sensors such as radar, LIDAR, cameras, incremental counters (measuring wheel rotation to perform an "odometric" measurement), and ultrasound transceivers are usually precise.

Nevertheless, the exclusive use of relative sensors, even though the initial position is known with certitude, always leads to unbounded incertitude. This is known as the "dead-reckoning drifting problem."

Fusing absolute and relative sensors is a means of getting absolute and precise location. The Kalman filter (KF) theory gives excellent results, since this estimator is optimal [15]. For each step, an a priori prediction of the mobile's position distribution, called belief function, is computed from past measurements and beliefs, and actuator commands and inner self-representations (system model and sensor model). When a new set of sensor data is collected, the prediction is corrected into a posterior belief function for the next cycle. This explains why KFs are often categorized as predictor–corrector state estimator filters.

In common modern navigation applications, when 10 m precision GPS and odometry are fused, they can provide estimations that are close to 1 m precision and are thus suitable for advisory navigation systems. Use of expensive sensors or correction services (which imply some form of communication) can boost the precision up to 10 cm and are thus are suitable for vehicle control systems [16].

3.2.3 Wireless Network Technologies

Wireless communication technologies applied to the automotive world bring forth new applications such as navigation, fleet management, billing facilities, and road security. All the applications currently brought to the market rely on vehicle-to-infrastructure (V2I) data exchange with private networks or protocols. Standards for vehicle-to-vehicle (V2V) are now being studied. Given that vehicles count by millions and have a life expectancy of 10 years or more, and also that technologies are in constant evolution, the communication system and its associated protocols have to match important reliability, scalability, and flexibility requirements. Thus, seven-layer open system interconnection (OSI) compliance will be an important issue for the interoperability of future V2V communication technologies.

IPv6 routing (third OSI layer) may play an important role, since this protocol addresses both wireless networks such as Wifi (IEEE 802.11), Wimax (IEEE 802.16), and cellular networks (global system for mobile [GSM] communications, general packet radio service [GPRS], universal mobile telecommunications system [UMTS]), and simplifies the routing of data through heterogeneous nodes, while reducing overhead, since this protocol is "physical layer agnostic." It also provides routing protocols compatible with network mobility, as seen in the works of International Engineering Task Force for Network Mobility (IETF NEMO), and compatible with ad-hoc networks, as seen in the works of IETF Mobile Ad-hoc Networking (MANET) [17]. Quality of service will be an important issue in vehicular communications, since vehicle density peaks, such as those met during traffic congestions, can lead to digital network traffic congestions. Appropriate strategies are needed so that priorities are handled properly and essential information is still diffused.

Integration of these communication technologies is now dealt with more intensively through projects such as CVIS [18] and SafeSpot [19], aiming at improving traffic efficiency and safety while reducing pollution as well as designing more efficient fleet management systems.

3.2.4 Intelligent Control Applications

Traditional actuators carry out the driver's commands in a direct manner, whereas electronic control units (ECUs) can enhance them beyond human reach. The human actuation bandwidth, for which 10 Hz can be considered as an upper bound, is easily outdone by electromechanical devices. The next example shows how "intelligent control" can greatly enhance the output of braking systems.

3.2.4.1 Antilock Braking System

Antilock braking system (ABS) for automotive applications was introduced by Teldix in the 1960s, before Bosch bought the patent and enhanced it through successive generations of ABSs. The system can activate and release each brake up to 50 times/s. This fine control enables each tire to get optimal longitudinal and satisfactory lateral frictions, so that braking distance is minimized while maneuverability stays acceptable.

Nearly optimal emergency braking is obtained by staying close to the maximum friction force possible, without going beyond it since that means entering an unstable region that leads to a blocked wheel state. ABS control algorithms guarantee that the functioning point stays between boundaries A and B shown in Figure 3.5. When brakes are actuated, vehicle dynamics and friction law quickly slide the operating

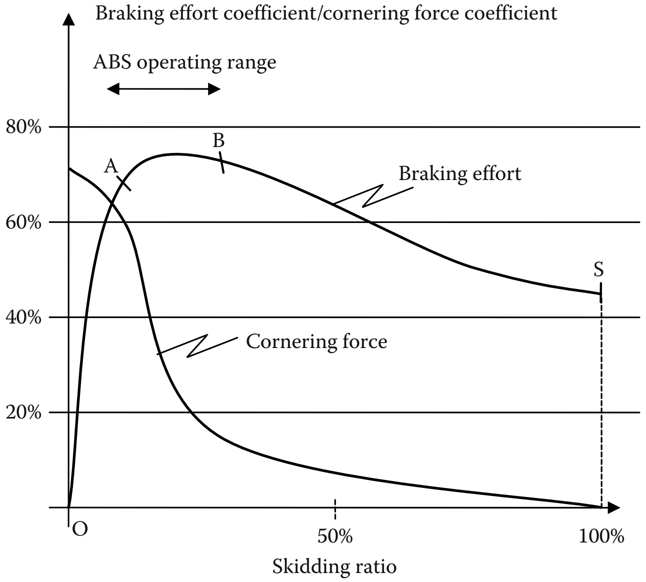


FIGURE 3.5 Efficiency of braking.

point from point O to S, leading to a final 50% braking effort coefficient and 100% skidding ratio (the wheel is blocked). If the brakes are released, skidding is progressively reduced to zero again. If sufficient actuation reversion rate is available, the braking effort can be maintained at a high rate.

An alternative to the ABS is to control the pressure on the brake pedal with continuous laws instead of bang–bang actuation, as brake-by-wire systems do.

3.2.4.2 Electronic Stability Program

As an extension of ABS, electronic stability program (ESP) systems put optimal yaw (spin) control at stake, introducing a correlation between the steering angle, individual wheel speed, and engine power to optimize trajectory control in bends, avoiding understeering (plowing) and oversteering (fishtailing) by fine-tuning individual wheel rotation speed. The increase in vehicle controllability overwhelms human capability, and it can only be done with numerical technologies, such as internal state representation and estimation.

3.2.4.3 X-by-Wire Technologies

X-by-wire technologies is a generic term designating electronic architectures that suppress traditional mechanical and hydraulic actuators and intermediate components and replace them with electronic and electric components, which are linked together only by wires, used as informational and power vectors. X-by-wire technologies open the possibility for enhanced comfort, as well as enhanced control applications and even push vehicle performance forward, since they introduce all-electronic actuator control, at the possible expense of dependability. The X-by-wire consortium has been in charge of defining standard architectures, and promoting the benefits of these technologies since the mid-1990s [20,21].

The SPARC X-by-wire prototypes, developed during the SPARC project [3,22], extend and merge classical ABS and ESP functions, in a global vehicle controller, that coordinates brake and steering actuation for increased system commandability. Dependability issues are addressed through the systematic use of dual components (quadrupled in the case of ECUs), as well as FlexRay deterministic and fault-tolerant multiplexed bus.

3.2.5 Latest Driving Assistance

Through the use of sensors and with the help of actuators that can be controlled by ECUs, numerous forms of assistance are now coming into the vehicle. The very first driver's assistance that arrived in the automobile was the antiblocking system for the wheels (often called ABS, which is a brand owned by Bosch). Although numerous mechanical and hydraulic systems were tested, it was the arrival of electronic sensors and control units that solved the problem in the early 1960s.

From then until the late 1990s, a few new functions have been introduced to control the vehicle to improve the safety and the comfort. However, recently, new systems have appeared for controlling the speed and/or the steering in various situations.

One of the first to appear was what is now called ACC. This is an improvement of the standard cruise control, which is a regulation of the vehicle speed to a set value. The ACC also tries to maintain the set speed but in the case of a slower vehicle in the same lane, the regulation is then set on the distance in relation to this vehicle. Through the use of a distance and/or relative speed sensor (either LIDAR or radar), a control unit regulates the speed through actions on the engine power or on the brakes. This technique was first developed and demonstrated in the European Prometheus project [23]. The first systems that appeared on the market (in Japan) in the late 1990s operated poorly in bad weather (due to the use of LIDAR) and in turn because the target vehicle was lost by the sensor. New systems, based on radar and using rotation information or vision to take into account the turning radius of the road have much better performance and even feature a "stop-and-go" function that is an essential application in crowded environments. The latest function using similar technology is advanced obstacle detection and precrash detection thanks to a fusion of information between radar and a stereo camera [24].

New functions now concern the lateral guidance of the vehicle with electric steering actuators and an electronic controller. One of the first applications to hit the market is lane-keeping, where information about the vehicle position in the lane is obtained through image processing. Using this position, an error is calculated and torque is applied on the steering column in order to bring the vehicle to the center of the lane. For the moment, this is only to assist the driver, who is responsible for steering the vehicle, but he or she feels secure that the car automatically wants to stay in the middle of the lane [24].

Using the same electric steering actuators and vision sensors that can reconstruct the 3D space around the vehicle, a more spectacular application is now being introduced in the most advanced vehicles: parking assistance. Using the 3D information obtained from one or several cameras inside the vehicle, complemented by ultrasonic sensors, a computer generates the optimal trajectory to perform a parallel parking maneuver with the steering. The driver has just to control the speed of his or her vehicle [24].

3.3 Dependability Issues

3.3.1 Introduction

Dependability is a generic term associated with functional and dysfunctional properties of a system or subsystem. Dependability attributes are most frequently referred to by their acronym—RAMS: "reliability," which designates the continuity of a system's service and is often measured by its mean time between failures (MTBF); "availability," which designates a system's readiness for service; "maintainability," which designates a system's ability to recover from a failure and is often measured by its mean time to repair (MTBR); and "safety," which designates the risk of catastrophic (lethal) failure, a combination of probability and gravity. "Security" may also be mentioned, being a system's ability to authorize known users to operate it and resist malicious attacks.

Certification and homologation procedures will guarantee that intelligent vehicles operating on public roads have attributes within acceptable boundaries. Those norms are under permanent evolution, integrating more and more mandatory requirements such as safety belts (1979), electronic immobilizers (1995), and ABS (2003), as far as European standards are concerned.

Manufacturers and suppliers have carried out intensive studies on hardware and software architecture standards, as seen in the emergence of MISRA, automotive open system architecture (AUTOSAR), X-by-wire consortiums [20,25,26], which define guidelines and standards for high quality safety-related system design. In the case of drive-by-wire technology, those guidelines may become new certification standards when sufficient proof of safety in the absence of mechanical intrinsic security is accepted by authorities.

However, most of the time, existing certification standards are tolerant toward automotive innovations. Obstacles for innovation can be found in the liability issue, consumer acceptance, and sometimes in mismatched homologation-related requirements. The liability issue puts manufacturer/supplier/customer potential struggles at stake. Driver liability is usually engaged unless functional safety problems—system malfunctions—are met and proved. Data-logging devices are an effective way of solving upcoming liability discussions and have already been used since the introduction of air bag systems [27,28].

3.3.2 Fail-Safe Automotive Transportation Systems

Efficient transportation systems imply important kinetic energy, and potential catastrophic (lethal) accidents. The root cause of an accident can be extrinsic or intrinsic to the transportation system and its users. Overall safety level is achieved through risk reduction analyses that are performed at different levels to prevent disastrous events from occurring:

1. Improve intrinsic system reliability (harden hardware and software components)
2. Exclude potential threats from the system's operational infrastructure
3. Define appropriate operating procedures for preventing users' misuse of the system

Among these levels, automotive applications usually have fewer solutions than rail or air transportation systems, since

1. Cost and space constraints restrict the use of redundancy
2. Road infrastructure is fully open
3. Most drivers are not "professionals"

A transportation system is called "fail-safe" when the occurrence of a failure leads to a safe state, where restricted transportation service is allowed. In the case of a rail transportation system, a safe state is reached by stopping the train. Global system design prevents other trains from entering the bloc (which is greater than the maximal stopping distance behind a stopped vehicle).

In the case of air transport, failures are handled through redundancy. Any critical component is tripled or quadrupled and a failed component is taken out of the control loop by a voting mechanism.

In the case of automotive transportation systems, experiencing a breakdown is more complicated. Nevertheless, from the other cars' points of view, a disabled or stopped car is an obstacle. In other words, appropriate fault handling at the vehicle level is not always sufficient to eradicate the problem, and the latter can have echoes at a fleet level.

Several advanced driver assistance systems (ADAS) are being studied and brought to the market in order to detect obstacles and assist collision avoidance maneuvers in the case of disabled vehicles on the road. When, despite warnings, the vehicle enters the inevitable collision state (ICS) [29], and before collision occurs, suitable actions can help and mitigate damage before "passive security" takes the last stand. Inevitable collision detection, automatic braking, seatbelt pretension, anticipated air bag firing are some of the challenges facing intelligent vehicle applications today. Entering ICS can result from an obstacle "moving" into the vehicle's danger zone, such as a tree falling on the road or another car behaving wildly. A vehicle can also enter ICS because it has an insufficient perception of the actual time–space configuration, maybe due to sensor failure or because the obstacle is hidden behind a curve.

V2V and V2I communication technologies bring forth new applications, enhancing security and providing solutions to prevent other vehicles from entering ICS:

1. A stopped car can broadcast—or "roadcast"—a warning signal to all vehicles within range.
2. The accident can be uploaded to appropriate traffic monitoring central systems, and downloaded again toward vehicles entering the vicinity.
3. By extension, if sufficient bandwidth is available, every vehicle should roadcast their trajectories to offer redundant environmental perception.

These applications have strong potential in terms of security improvement since they are working ahead of ICS occurrence and working on root causes rather than trying to mitigate effects, with "perceptive" ranges that can go far beyond the range of natural or technological sensors. In that sense, communication technologies bend new corners in automotive safety issues. Moreover, they provide a global approach. The "fail-safe automotive transportation system" being designed is now about to reach fleet scale. Its overall safety level will depend on sensor technologies, the dependability of controllers and actuators, as well as the efficiency of information propagation throughout the whole fleet, as seen in the CityMobil research project [30].

3.3.3 Intelligent Autodiagnostic

Fault-detection aptitude is essential for the survivability of a transportation system and its users. The first traditional application of fault detection is the implantation of an adequate sensor to monitor physical parameters that give an overview of the system's and the subsystem's health. Tire pressure or motor temperature sensors are good examples of straightforward monitoring and diagnostic applications.

Common fault-detection and diagnostic applications are case-based: simple rules are designed to decide whether a fault occurred or not, using threshold, 2D domains, etc. Once a fault is detected, the fault is logged inside the ECU, and according to its gravity, an alert may be immediately displayed to the driver.

As a response to the increase in software preponderant automotive systems, one also has to detect execution faults triggered by external factors (weather, electromagnetic interaction, and wire rupture). In order to ensure that the commands of an ECU are emitted and received the way they are meant to be, it is possible to introduce redundancy, which can be informational or hardware, so that inconsistency is detected and appropriate fallback behaviors are triggered. In most of the cases, the cost, space, and time to market constraints make full-ECU redundancy inaccessible in most automotive applications. Thus, redundancy is usually achieved inside a single ECU. When inputs are concerned, the receiving ECU can check the consistency of redundant signals and activate appropriate procedures when they do not match (e.g., log the default, alert the driver, stop the activity of the (sub)system). When outputs are concerned, a feedback mechanism is redirected toward microprocessor inputs. Inconsistency between commanded output and measured output means either output or feedback loop failure. Whatever the cause, which could only be determined through the introduction of additional (and failure-prone) observation points, the overall output device is considered as defective.

More complex diagnostic applications are experimented in the field, such as rupture detection, which uses identification techniques to estimate and monitor parameters that are not directly observable, such as suspension damping and stiffness coefficients, in order to detect imminent mechanical breakdowns with model-based fault-detection algorithms [31].

Some research is conducted in the field of wired network diagnosis, such as in the Smart Embedded Electronic Diagnosis System (SEEDS) project, where embedded reflectometry chips are used to monitor the condition of an embedded harness. The latter can represent up to 4 km of wires and is a major source of system failures [32].

Since embedded functions are usually distributed over several ECUs, it can be difficult to determine which ECU is the cause of a function failure. On the other hand, ECUs usually participate in several functions, and a faulty ECU can imply more than one function-level failure. Introducing the correlation between different defaults can be an effective means of improving diagnostic, switching from a local to a system point of view; fuzzy logic and neural networks may help to develop such system-level diagnostics [33].

3.4 Fully Autonomous Car: Dream or Reality?

In order to meet the challenges of the twenty-first century in terms of mobility (for people and goods), a new form of transport needs to be developed. We believe that the new transport technologies will be mostly based on automated road transport using existing and new road infrastructures. The key points will be a hierarchical network with very high capacity links, optimization, and demand control. The objectives are to provide door-to-door transport to anyone (or anything), at any time but at different costs depending on the modes used. These technologies are now being developed through several European programs such as CyberCars and CityMobil (see www.cybercars.org and www.citymobil-project.eu). Several systems are now at the implementation level following European level recommendations [34,35]. This section tries to explore the long-term future of these systems.

3.4.1 Automated Road Vehicles

Automated road transport will include various types of vehicles, as is the case with today's manual vehicles on the existing road infrastructure. The goal is to optimize the efficiency of the system. Therefore, depending on the demand on a particular road link, the users will be encouraged (possibly through pricing) to take a high-capacity vehicle. This will probably mean a change in vehicle since high-capacity vehicles will not do door-to-door but will run on specific routes. These high-capacity vehicles will be like buses (50–100 passengers), but these buses will have the capacity to form closely linked platoons with a very high combined capacity (similar to a suburban train capacity with properly designed platforms). The top speed of these high-capacity vehicles should be on the order of 100 km/h in city environments. For intercity travel, very high speed vehicles could be considered (200 km/h). Early models of such vehicles are already in operation (Phileas, CVIS, IMTS, etc., see Figure 3.6) and are derived from the bus rapid transit (BRT) concept with advanced technologies for lateral and sometimes longitudinal guidance to improve their operation. It must be pointed out that BRT has been suggested in a recent study to be the "best option" for reducing GHG in developing cities [36].

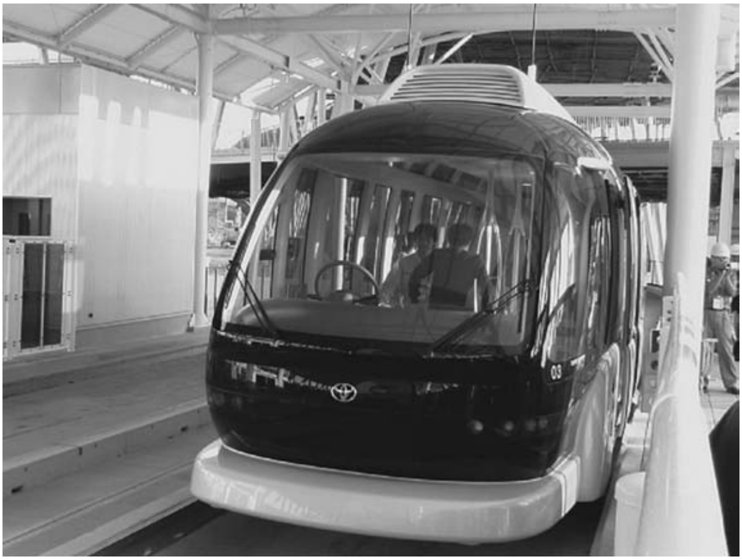


FIGURE 3.6 IMTS from Toyota.

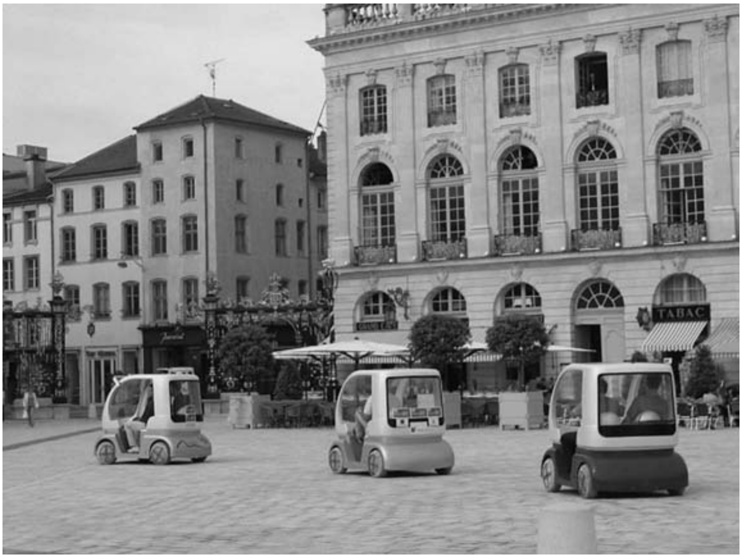


FIGURE 3.7 CyCabs from Robosoft.

For goods transport we should have vehicles of similar size, designed for the transport of standard containers.

The second type of vehicles will be individual ones, varying in size from 1 to 10 passengers and they can be private or public (although the need for a private vehicle will be very limited since the service in a public one should be very similar and much cheaper). These cybercars (Figure 3.7), as they are now called, will operate on demand (including the selection of vehicle type) for passengers going from one location to their destination or to a high capacity or high speed link (such as a train station or an airport) with no intermediate stop. Similar-sized vehicles will be used for delivering goods door-to-door, and could possibly even collect the garbage, using specific containers of various standard sizes. The handling of these containers would be automatic. This trend follows from the development of car-sharing schemes [37] and electric vehicle goods distribution in cities [38].

3.4.2 Automated Road Network

The automated road network (ARN) vehicles will use a combination of roads of different types as is the case today with traditional vehicles. The objective is to have a network with different capacities and speeds as required to optimize trip time and capacity at a given cost.

The capillaries of the network will be today's existing streets, which might be redesigned slightly to improve the urban esthetics and pedestrian space. Some space will be made available for temporary parking for loading and unloading passengers or goods. The individual automated road vehicles will use these capillaries at low speed

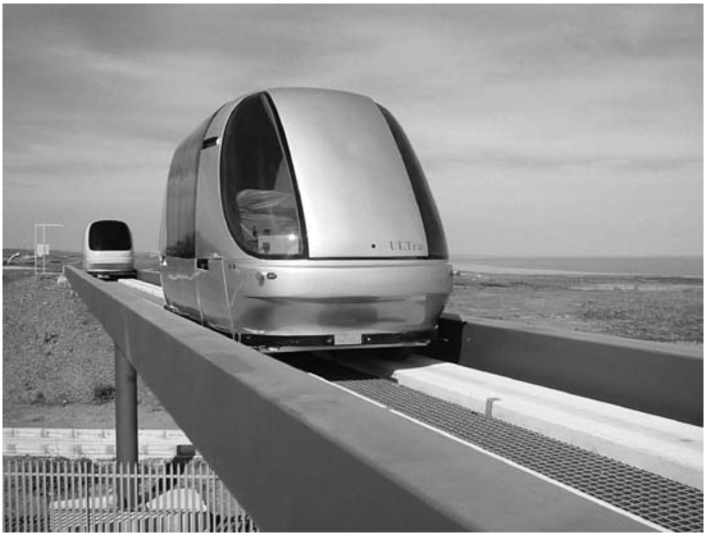


FIGURE 3.8 ULTra track.

(less than 30 km/h) since the space might be shared with pedestrians and cyclists and a few manually driven vehicles.

The next level of roads will be today's arteries and they will be designed for high capacity. They will be used by high-capacity vehicles as well as individual ones, although high-capacity vehicles will be given a priority at peak time (through pricing). These arteries will be redesigned to ensure the best possible speed at the lowest risk, as done today for light rail or BRT. In some cases, they might be completely separated from pedestrians (Figure 3.8). Stations (and exits for individual automated vehicles) should be fairly widely spaced to ensure high speed, since local, individual vehicles should be available for the last (or first) leg of the trip. At the stations, the passengers would move very conveniently from a collective vehicle to an individual one and vice versa.

The last level of the ARN will consist of a new infrastructure built specifically for the automated vehicles. It would be a light infrastructure, completely segregated but with entry and exit points to the other two levels of the ARN. This infrastructure would be mostly above ground as is constructed today for personal rapid transit (PRT). Today's freeways would be converted totally (or partially, preserving a single lane, if manual vehicles are still allowed) into ARNs, with the stations located at some of today's exchanges.

3.4.3 Automated Road Management

For the proper operation of automated road transport, it is essential that all the elements of the system be properly managed in order to obtain the best efficiency and service to the user. The general principle is to satisfy most of the demand while minimizing cost. Since transport demand can exceed the capacity at particular times, some form of demand management must be applied. In a liberal society, it is likely that this control will be done through some form of trip pricing.

Since it will be more efficient to move individuals (or goods) in large quantities simultaneously, mass transport will be encouraged as long as the demand can justify the use of large vehicles. If this is not the case, only individual vehicles will be offered. The automated road management will therefore have to manage the pricing structure, possibly through companies that will provide the services and the individual billing of the customers (several companies might run their automated vehicles through the same network).

The management center will also manage the navigation of each automated vehicle so that the best route is used for a particular trip. The best route is not necessarily the fastest one but the one that optimizes the entire system. The management center has also to manage other resources such as loading/unloading zones, standby locations (parking), energy, and vehicle maintenance facilities [39]. Therefore, the management will seek to reach an optimum with respect to certain criteria that will have to be decided by the operators in conjunction with the political level.

Finally, the management will have to manage the maintenance and the evolution of the system as well as the unavoidable emergencies (disabled vehicles, broken infrastructure, objects on the roads, etc.).

3.4.4 Deployment Paths

Obviously the deployment of the automated road transport will not happen everywhere at the same time. There are three trends emerging toward this future scenario. One is driving assistance that has been spreading quite rapidly since the late 1990s. As we saw previously, numerous techniques have appeared in recent high-end private vehicles, such as longitudinal control using radar and lateral control using vision techniques [40]. The first vehicles incorporating these technologies are now on the market and it is possible that in 2030, 50% of the vehicles manufactured will incorporate them (while low-cost "manual" vehicles will be mostly reserved for developing countries).

Such techniques are now also appearing on buses and can lead the way to the first generation of large automated road vehicles with the concept of automated BRT (ABRT) as a cheaper alternative to automated metros. Several manufacturers are already addressing this market.

The second trend is the arrival of people-movers based on automated guided vehicles in specific locations and on dedicated tracks (protected or not). These systems are now starting to be deployed, the biggest hurdle being the lack of proper legislation that prevents the operation of totally automated vehicles on public roads at the moment. This problem is now being addressed at the European level.

The third trend is the rapid development (mostly in large cities) of car-sharing schemes, addressing the needs of urbanites who seek a complement to public transport and punctual needs for a private transport. Several cities and states are now encouraging this trend which can lead to reduced needs for parking space as well as cleaner, safer vehicles. In 2006, a French law was passed to facilitate the deployment of these services (Ries law).

It can be forecast that in the next 10 years, these last two trends will merge with individual vehicles with dual-mode capabilities: manual (assisted) driving on regular roads and fully automatic driving in dedicated zones where no (or few) manual vehicles will be allowed, therefore ensuring smooth and safe operation of the automated vehicles.

3.5 Conclusion

As we have seen, control technologies based on sophisticated sensors and control units running advanced algorithms are now quickly arriving in standard road vehicles, that is, passenger cars, buses, trucks, and vans. These technologies are being developed for the moment as comfort features but are now increasingly impacting the safety and efficiency of the vehicles.

These techniques, associated with communications and global control of the road network, will evolve toward a completely new form of transport, which will rely less and less on the human driver. Perhaps in 50 years, manually driven vehicles will be left for a few enthusiasts as is the horse nowadays.

References

1. ACEA. European Automobile Manufacturers Association Web Site (www.acea.be).
2. Swedish National Road Administration, *Vision Zero—From Concept to Action*. SNRA, Borlange, 1999.
3. PReVENT Website: http://www.prevent-ip.org.
4. *Evaluation of Scenarios to Deployment of ADAS/AVG Systems in Urban Contexts* Deliverable D11, Stardust Project (www.trg.soton.ac.uk/stardust/index.htm).
5. Ieromonachou, P., Potter, S., and Warren, J., *Comparing Urban Road Pricing Implementation and Management Strategies from the UK and Norway*. Faculty of Technology, Open University, 2005.
6. Lowson, M., Energy use and sustainability of transport systems. Advanced Transport Group, University of Bristol, UK, Deliverable of CyberCars Contract. (www.cybercars.org).
7. Fox, D. et al., Position estimation for mobile robots in dynamic environments, in *Proceedings of the 15th National Conference on Artificial Intelligence*, Madison, WI, 1998.
8. Shoval, S. and Borenstein, J., Measurement of angular position of a mobile robot using ultrasonic sensors, in *Proceedings of the ANS Conference on Robotics and Remote Systems*, Pittsburgh, PA, April 1999.
9. Thomas, K., Laserscanner for automotive applications, in *Workshop on Intelligent Transportation*, Germany, 2004.
10. Evans, D., *Performance Analysis of Next-Generation LADAR for Manufacturing*, NISTIR 7117, Building and Fire Research Laboratory National Institute of Standards and Technology, Gaithersburg, MD, May 2004.
11. Honma, S. and Uehara, N., Millimeter-wave radar technology for automotive application, Technical Reports, Mitsubishi Electric ADVANCE, June 2001.

12. Hoess, A. et al., Design and realization of a novel, synchronized 77 GHz radar network for automotive use, in *IMS Workshop on Circuit and Antenna Technologies for Automotive Radars*, Seattle, June 3, 2002.
13. Schauerte, J. et al., A 360×226 pixel CMOS imager chip optimized for automotive vision applications Frank, Delphi Research Labs, SAE technical paper 2001-01-0317.
14. Muramatsu, S. et al., Automotive vision platform equipped with dedicated image processor for multiple applications, SAE technical paper 2004-01-0179.
15. Welch, G. and Bishop, G., An introduction to the Kalman filter, Technical report, University of North Carolina, Chapel Hill, NC, 1995.
16. El Najjar, M.E. and Bonnifait, Ph., A roadmap matching method for precise vehicle localization using belief theory and Kalman filtering, in *Proceedings of the 11th International Conference on Advanced Robotics*, IEEE, Portugal, June 2003.
17. Ernst, T. and De La Fortelle, A., Car-to-car and car-to-infrastructure communication based on Nemo and Manet in IPv6, in *Proceedings of the 13th World Congress on Intelligent Transport Systems and Services*, London, October 2006.
18. CVIS Website: http://www.cvisproject.org.
19. SafeSpot Website: http://www.safespot-eu.org.
20. *Safety Related Fault Tolerant Systems*—X-by-Wire consortium final report, 1998.
21. Whitfield, K., Solve for X: X-by-wire technologies, *Automotive Design & Production*, December 2001.
22. SPARC Website: http://www.sparc-eu.net/.
23. EurekaPrometheusProject.http://en.wikipedia.org/wiki/EUREKA_Prometheus_Project.
24. Lexus Web Site: http://www.lexus-europe.com/about/pursuit-of-perfection/index.aspx (see LS460's advanced obstacle detection, lane-keeping assist and intelligent parking assist).
25. MISRA Web site: http://www.misra.org.uk/.
26. AUTOSAR, Automotive Open System Architecture (http://www. autosar.org).
27. Blum, J.J. and Eskandarian, A., Managing effectiveness and acceptability in intelligent speed adaptation systems, in *Proceedings of the IEEE ITSC 2006*, Toronto, Canada, September 17–20, 2006.
28. van Wees, K., Liability aspects of ISA, in *12th World Congress on ITS*, San Francisco, November 6–10, 2005.
29. Fraichard, T. and Asama, H., Inevitable collision states. A step towards safer robots? *Advanced Robotics*, 18(10):1001–1024, 2004.
30. CityMobil Website: http://www.citymobil-project.eu/.
31. Börner, M., Zele, M., and Isermann, R., Comparison of different fault detection algorithms for active body control components: Automotive suspension systems, in *Proceedings of the American Control Conference*, Arlington, VA, June 2001.
32. Auzanneau, F., Olivas, M., and Ravot, N., A simple and accurate model for wire diagnosis using reflectometry, in *Progress in Electromagnetics Research Symposium 2007*, Prague, Czech Republic, August 27–30, 2007.
33. Gusikhi, O., Rychtyckyj, N., and Filev, D., Intelligent systems in the automotive industry: Applications and trends, *Knowledge and Information Systems*, 12(2), July 2007.
34. EPC Task Force on Transport, 12 Prescriptions for a European sustainable mobility policy, EPC Working Paper No. 16, The EPC Task Force on Transport, 2005.

35. Towards a thematic strategy on the urban environment, Communication from the EC to the Parliament, 2004.
36. Vincent, W. and Jerram, L.C., The potential for bus rapid transit to reduce transportation-related CO_2 emissions, *Journal of Public Transportation*, 2006 (BRT Special Edition).
37. Parent Michel and Texier Pierre-Yves, A public transport system based on light electric cars, in *Fourth International Conference on Automated People Movers*, Irving, March 1993.
38. ELCIDIS, Electric Vehicle City Distribution, Elcidis Final Report TR 0048/97, European Commission, 2002.
39. Awasthi, A., Benabid, S., Talamona, A., and Parent, M., Centralized fleet management for cybernetic transportation system, in *Proceedings of ITS 2003*, Madrid, October 2003.
40. Parent Michel and Blosseville Jean-Marc., Automated vehicles in cities: A first step towards the automated highway, in *SAE Future Transportation Technology Conference*, Costa Mesa, CA, August 11–13, 1998.

II

Embedded Communications

4

A Review of Embedded Automotive Protocols

Nicolas Navet
National Institute for Research in Computer Science and Control
RealTime-at-Work

Françoise Simonot-Lion
Lorraine Laboratory of Computer Science Research and Applications

4.1 Automotive Communication Systems: Characteristics and Constraints

4.1.1 From Point-to-Point to Multiplexed Communications

Since the 1970s, there has been an exponential increase in the number of electronic systems that have gradually replaced those that are purely mechanical or hydraulic. The growing performance and reliability of hardware components and the possibilities brought about by software technologies enabled implementing complex functions that improve the comfort of the vehicle's occupants as well as their safety. In particular, one of the main purposes of electronic systems is to assist the driver to control the vehicle through functions related to the steering, traction (i.e., control of the driving torque), or braking such as the antilock braking system (ABS), electronic stability program (ESP), electric power steering (EPS), active suspensions, or engine control. Another reason for using electronic systems is to control devices in the body of a vehicle such as lights, wipers, doors, windows, and, recently, entertainment and communication equipments (e.g., radio, DVD, hands-free phones, and navigation systems).

In the early days of automotive electronics, each new function was implemented as a stand-alone electronic control unit (ECU), which is a subsystem composed of a microcontroller and a set of sensors and actuators. This approach quickly proved to be insufficient with the need for functions to be distributed over several ECUs and the need for information exchanges among functions. For example, the vehicle speed estimated by the engine controller or by wheel rotation sensors has to be known in order to adapt the steering effort, to control the suspension, or simply to choose the right wiping speed. In today's luxury cars, up to 2500 signals (i.e., elementary information such as the speed of the vehicle) are exchanged by up to 70 ECUs [1]. Until the beginning of the 1990s, data were exchanged through point-to-point links between ECUs. However this strategy, which required an amount of communication channels of the order of n^2 where n is the number of ECUs (i.e., if each node is interconnected with all the others, the number of links grows in the square of n), was unable to cope with the increasing use of ECUs due to the problems of weight, cost, complexity, and reliability induced by the wires and the connectors. These issues motivated the use of networks where the communications are multiplexed over a shared medium, which consequently required defining rules—protocols—for managing communications and, in particular, for granting bus access. It was mentioned in a 1998 press release (quoted in Ref. [2]) that the replacement of a "wiring harness with local area networks (LANs) in the four doors of a BMW reduced the weight by 15 kilograms." In the mid-1980s, the third part supplier Bosch developed controller area network (CAN), which was first integrated in Mercedes production cars in the early 1990s. Today, it has become the most widely used network in automotive systems and it is estimated [3] that the number of CAN nodes sold per year is currently around 400 million (all application fields). Other communication networks, providing different services, are now being integrated in automotive applications. A description of the major networks is given in Section 4.2.

4.1.2 Car Domains and Their Evolution

As all the functions embedded in cars do not have the same performance or safety needs, different quality of services (QoS) (e.g., response time, jitter, bandwidth, redundant communication channels for tolerating transmission errors, efficiency of the error detection mechanisms, etc.) are expected from the communication systems. Typically, an in-car embedded system is divided into several functional domains that correspond to different features and constraints (Chapter 1). Two of them are concerned specifically with real-time control and safety of the vehicle's behavior: the "power train" (i.e., control of engine and transmission) and the "chassis" (i.e., control of suspension, steering, and braking) domains. The third, the "body," mostly implements comfort functions. The "telematics" (i.e., integration of wireless communications, vehicle monitoring systems, and location devices), "multimedia," and "human–machine interface" (HMI) domains take advantage of the continuous progress in the field of multimedia and mobile communications. Finally, an emerging domain is concerned with the safety of the occupant.

The main function of the power train domain is controlling the engine. It is realized through several complex control laws with sampling periods of a magnitude of some milliseconds (due to the rotation speed of the engine) and implemented in microcontrollers with high computing power. In order to cope with the diversity of critical tasks to be treated, multitasking is required and stringent time constraints are imposed on the scheduling of the tasks. Furthermore, frequent data exchanges with other car domains, such as the chassis (e.g., ESP, ABS) and the body (e.g., dashboard, climate control), are required.

The chassis domain gathers functions such as ABS, ESP, automatic stability control (ASC), four-wheel drive (4WD), which control the chassis components according to steering/braking solicitations and driving conditions (ground surface, wind, etc.). Communication requirements for this domain are quite similar to those for the power train but, because they have a stronger impact on the vehicle's stability, agility, and dynamics, the chassis functions are more critical from a safety standpoint. Furthermore, the "X-by-Wire" technology, currently used for avionic systems, is now slowly being introduced to execute steering or braking functions. X-by-Wire is a generic term referring to the replacement of mechanical or hydraulic systems by fully electrical/electronic ones, which led and still leads to new design methods for developing them safely [4] and, in particular, for mastering the interferences between functions [5]. Chassis and power train functions operate mainly as closed-loop control systems and their implementation is moving toward a time-triggered approach [6–9], which facilitates composability (i.e., ability to integrate individually developed components) and deterministic real-time behavior of the system.

Dashboards, wipers, lights, doors, windows, seats, mirrors, climate control are increasingly controlled by software-based systems that make up the body domain. This domain is characterized by numerous functions that necessitate many exchanges of small pieces of information among themselves. Not all nodes require a large bandwidth, such as the one offered by CAN; this leads to the design of low-cost networks such as local interconnect network (LIN) and time-triggered protocol (TTP/A, Section 4.2). On these networks, only one node, termed the master, possesses an accurate clock and drives the communication by polling the other nodes, the slaves, periodically. The mixture of different communication needs inside the body domain leads to a hierarchical network architecture where integrated mechatronic subsystems based on low-cost networks are interconnected through a CAN backbone. The activation of body functions is mainly triggered by the driver and passengers' solicitations (e.g., opening a window, locking doors, etc.).

Telematics functions such as hand-free phones, car radio, CD, DVD, in-car navigation systems, rear seat entertainment, remote vehicle diagnostic, etc., are becoming more and more numerous. These functions require a lot of data to be exchanged within the vehicle but also with the external world through the use of wireless technology (see, for instance, Ref. [10]). Here, the emphasis shifts from messages and tasks subject to stringent deadline constraints to multimedia data streams, bandwidth sharing, and multimedia QoS where preserving the integrity (i.e., ensuring that information will not be accidentally or maliciously altered) and confidentiality of information is crucial. HMI aims to provide interfaces that are easy to use and that limit the risk of driver inattention [11].

Electronic-based systems for ensuring the safety of the occupants are increasingly embedded in vehicles. Examples of such systems are impact and roll-over sensors, deployment of air bags and belt pretensioners, tyre pressure monitoring, and adaptive cruise control (ACC)—the car's speed is adjusted to maintain a safe distance with the car ahead. These functions form an emerging domain usually referred to as "active and passive safety."

4.1.3 Different Networks for Different Requirements

The steadily increasing need for bandwidth* and the diversification of performance, costs, and dependability† requirements lead to a diversification of the networks used throughout the car. In 1994, the Society for Automotive Engineers (SAE) defined a classification for automotive communication protocols [13–15] based on data transmission speed and functions that are distributed over the network. Class A networks have a data rate lower than 10 kbps and are used to transmit simple control data with low-cost technology. They are mainly integrated in the body domain (seat control, door lock, lighting, trunk release, rain sensor, etc.). Examples of class A networks are LIN [16,17] and TTP/A [18]. Class B networks are dedicated to supporting data exchanges between ECUs in order to reduce the number of sensors by sharing information. They operate from 10 to 125 kbps. The J1850 [19] and low-speed CAN [20] are the main representations of this class. Applications that need high-speed real-time communications require class C networks (speed of 125 kbps–1 Mbps) or class D networks‡ (speed over 1 Mbps). Class C networks, such as high-speed CAN [21], are used for the power train and currently for the chassis domains, while class D networks are devoted to multimedia data (e.g., media-oriented system transport, MOST [22]) and safety critical applications that need predictability and fault-tolerance (e.g., TTP/C [23] or FlexRay [24] networks) or serve as gateways between subsystems [25].

It is common, in today's vehicles, that the electronic architecture include four different types of networks interconnected by gateways. For example, the Volvo XC90 [3] embeds up to 40 ECUs interconnected by a LIN bus, a MOST bus, a low-speed CAN, and a high-speed CAN. In the near future, it is possible that a bus dedicated to occupant safety systems (e.g., air bag deployment, crash sensing), such as the "Safe-by-Wire plus" [26], will be added.

* For instance, in Ref. [5], the average bandwidth needed for the engine and the chassis control is estimated to reach 1500 kbps in 2008 while it was 765 kbps in 2004 and 122 kbps in 1994.

† Dependability is usually defined as the ability to deliver a service that can justifiably be trusted, see Ref. [12] for more details.

‡ Class D is not formally defined but it is generally considered that networks over 1 Mbps belong to class D.

4.1.4 Event-Triggered versus Time-Triggered

One of the main objectives of the design step of an in-vehicle embedded system is to ensure a proper execution of the vehicle functions, with a predefined level of safety, in the normal functioning mode but also when some components fail (e.g., reboot of an ECU) or when the environment of the vehicle creates perturbations (e.g., electromagnatic interference [EMI] causing frames to be corrupted). Networks play a central role in maintaining the embedded systems in a "safe" state since most critical functions are now distributed and need to communicate. Thus, the different communication systems have to be analyzed with regard to this objective; in particular, messages transmitted on the bus must meet their real-time constraints, which mainly consist of bounded response times and bounded jitters.

There are two main paradigms for communications in automotive systems: event-triggered and time-triggered. Event-triggered means that messages are transmitted to signal the occurrence of significant events (e.g., a door has been closed). In this case, the system possesses the ability to take into account, as quickly as possible, any asynchronous events such as an alarm. The communication protocol must define a policy to grant access to the bus in order to avoid collisions; for instance, the strategy used in CAN (Section 4.2.1.1) is to assign a priority to each frame and to give the bus access to the highest priority frame. Event-triggered communication is very efficient in terms of bandwidth usage since only necessary messages are transmitted. Furthermore, the evolution of the system without redesigning existing nodes is generally possible, which is important in the automotive industry where incremental design is the usual practice. However, verifying that temporal constraints are met is not obvious and the detection of node failures is problematic.

When communications are time-triggered (TT), frames are transmitted at predetermined points in time, which is well-suited for the periodic transmission of messages as it is required in distributed control loops. Each frame is scheduled for transmission at one predefined interval of time, usually termed a slot, and the schedule repeats itself indefinitely. This medium access strategy is referred to as time division multiple access (TDMA). As the frame scheduling is statically defined, the temporal behavior is fully predictable; thus, it is easy to check whether the timing constraints expressed on data exchanges are met. Another interesting property of TTPs is that missing messages are immediately identified; this can serve to detect, in a short and bounded amount of time, nodes that are presumably no longer operational. The first negative aspect is the inefficiency in terms of network utilization and response times with regard to the transmission of aperiodic messages (i.e., messages that are not transmitted in a periodic manner). A second drawback of TTPs is the lack of flexibility even if different schedules (corresponding to different functioning modes of the application) can be defined and switching from one mode to another is possible at runtime. Finally, the unplanned addition of a new transmitting node on the network induces changes in the message schedule and, thus, necessitates the update of all other nodes. TTP/C [23] is a purely TT network but there are networks, such as time-triggered CAN (TTCAN) [27], flexible time-triggered CAN (FTT-CAN) [28], and FlexRay, that can support a combination of both time-triggered and event-triggered transmissions.

This capability to convey both types of traffic fits in well with the automotive context since data for control loops as well as alarms and events have to be transmitted.

Several comparisons have been made between event-triggered and time-triggered approaches; the reader can refer to Refs. [1,28,29] for good starting points.

4.2 In-Car Embedded Networks

The different performance requirements throughout a vehicle, as well as competition among companies of the automotive industry, have led to the design of a large number of communication networks. The aim of this section is to give a description of the most representative networks for each main domain of utilization.

4.2.1 Priority Buses

To ensure at runtime the "freshness"* of the exchanged data and the timely delivery of commands to actuators, it is crucial that the medium access control (MAC) protocol is able to ensure bounded response times of frames. An efficient and conceptually simple MAC scheme that possesses this capability is the granting of bus access according to the priority of the messages (the reader can refer to Refs. [30,31] and Chapter 13 for how to compute bound on response times for priority buses). To this end, each message is assigned an identifier, unique to the whole system. This serves two purposes: giving priority for transmission (the lower the numerical value, the greater the priority) and allowing message filtering upon reception. The two main representatives of such "priority buses" are CAN and J1850.

4.2.1.1 CAN

CAN is without a doubt the most widely used in-vehicle network. It was designed by Bosch in the mid-1980s for multiplexing communication between ECUs in vehicles and thus for decreasing the overall wire harness: length of wires and number of dedicated wires (e.g., the number of wires has been reduced by 40%, from 635 to 370, in the Peugeot 307 that embeds two CAN buses with regard to the non-multiplexed Peugeot 306 [32]). Furthermore, it allows to share sensors among ECUs.

CAN on a twisted pair of copper wires became an ISO standard in 1994 [20,33] and is now a de facto standard in Europe for data transmission in automotive applications, due to its low cost, robustness, and bounded communication delays [3]. In today's car, CAN is used as an SAE class C network for real-time control in the power train and chassis domains (at 250 or 500 kbps), but it also serves as an SAE class B network for the electronics in the body domain, usually at a data rate of 125 kbps.

On CAN, data, possibly segmented in several frames, may be transmitted periodically, aperiodically, or on-demand (i.e., client/server paradigm). A CAN frame is

* The freshness property is verified if data have been produced recently enough to be safely consumed: the difference between the time when data is used and the last production time must always be smaller than a specified value.

labeled by an identifier, transmitted within the frame, whose numerical value determines the frame priority. CAN uses non-return-to-zero (NRZ) bit representation with a bit stuffing of length 5. In order not to lose the bit time (i.e., the time between the emission of two successive bits of the same frame), stations need to resynchronize periodically and this procedure requires edges on the signal. Bit stuffing is an encoding method that enables resynchronization when using NRZ bit representation where the signal level on the bus can remain constant over a longer period of time (e.g., transmission of "000000.."). Edges are generated into the outgoing bit stream in such a way as to avoid the transmission of more than a maximum number of consecutive equal-level bits (five for CAN). The receiver will apply the inverse procedure and de-stuff the frame. The standard CAN data frame (CAN 2.0A) can contain up to 8 bytes of data for an overall size of, at most, 135 bits, including all the protocol overheads such as the stuff bits. The reader interested in the details of the frame format and the bus access procedure should refer to Chapter 13. CAN bus access arbitration procedure relies on the fact that a sending node monitors the bus while transmitting. The signal must be able to propagate to the most remote node and return back before the bit value is decided. This requires the bit time to be at least twice as long as the propagation delay that limits the data rate; for instance, 1 Mbps is feasible on a 40 m bus at maximum while 250 kbps can be achieved over 250 m. To alleviate the data rate limit, and extend the life span of CAN further, car manufacturers are beginning to optimize the bandwidth usage by implementing "traffic shaping" strategies that are very beneficial in terms of response times; this is the subject of Chapter 14.

CAN has several mechanisms for error detection. For instance, it is checked that the cyclic redundancy check (CRC) transmitted in the frame is identical to the CRC computed at the receiver end, that the structure of the frame is valid and that no bit-stuffing error occurred. Each station that detects an error sends an "error flag," which is a particular type of frame composed of six consecutive dominant bits that allows all the stations on the bus to be aware of the transmission error. The corrupted frame automatically reenters into the next arbitration phase, which might lead it to miss its deadline due to the additional delay. The error recovery time, defined as the time from detecting an error until the possible start of a new frame, is 17–31 bit times. CAN possesses some fault-confinement mechanisms aimed at identifying permanent failures due to hardware dysfunctioning at the level of the microcontroller, communication controller, or physical layer. The scheme is based on error counters that are increased and decreased according to particular events (e.g., successful reception of a frame, reception of a corrupted frame, etc.). The relevance of the algorithms involved is questionable [34] but the main drawback is that a node has to diagnose itself, which can lead to the nondetection of some critical errors. For instance, a faulty oscillator can cause a node to transmit continuously a dominant bit, which is one manifestation of the "babbling idiot" fault (Chapter 6). Furthermore, other faults such as the partitioning of the network into several subnetworks may prevent all nodes from communicating due to bad signal reflection at the extremities. Without additional fault-tolerance facilities, CAN is not suited for safety-critical applications such as future X-by-Wire systems. For instance, a single node can perturb the functioning of the whole network by sending messages outside their specification (i.e., length and period of the frames). Many mechanisms were proposed for increasing the dependability of CAN-based networks (Chapter 6), but, if each proposal solves a particular problem, they have not necessarily been conceived to be combined. Furthermore, the fault-hypotheses used in the design of these mechanisms are not necessarily the same and the interactions between them remain to be studied in a formal way.

The CAN standard only defines the physical layer and data link layer (DLL). Several higher level protocols have been proposed, for instance, for standardizing startup procedures, implementing data segmentation, or sending periodic messages (see AUTOSAR and OSEK/VDX in Section 4.3). Other higher-level protocols standardize the content of messages in order to ease the interoperability between ECUs. This is the case for J1939, which is used, for instance, in Scania's trucks and buses [35].

4.2.1.2 VAN

Vehicle area network (VAN) [21] is very similar to CAN (e.g., frame format, data rate) but possesses some additional or different features that are advantageous from a technical point of view (e.g., no need for bit-stuffing, in-frame response: a node being asked for data answers in the same frame that contained the request). VAN was used for years in production cars by the French carmaker PSA Peugeot-Citroën in the body domain (e.g., for the 206 model) but, as it was not adopted by the market, it was abandoned in favor of CAN.

4.2.1.3 J1850 Network

The J1850 [19] is an SAE class B priority bus that was adopted in the United States for communications with nonstringent real-time requirements, such as the control of body electronics or diagnostics. Two variants of the J1850 are defined: a 10.4 kbps single-wire version and a 41.6 kbps two-wire version. The trend in new designs seems to be the replacement of J1850 by CAN or a low-cost network such as LIN (Section 4.2.3.1).

4.2.2 TT Networks

As discussed before, there are two types of communication networks: TT networks where activities are driven by the progress of time and event-triggered networks where activities are driven by the occurrence of events. Both types of networks have advantages but the TT bus is generally considered to be more dependable (refer, for instance, to Ref. [9] for a discussion on this topic). This explains that, currently, only TT communication systems are being considered for use in X-by-Wire applications. In this category, multiaccess protocols based on TDMA are particularly well suited; they provide deterministic access to the medium (the order of the transmissions is defined statically at the design time), and thus bounded response times. Moreover, their regular message transmissions can be used as "heartbeats" for detecting station failures. The three TDMA-based networks that could serve as gateways or for supporting safety critical applications are TTP/C [23], FlexRay (Section 4.2.2.1), and TTCAN (Section 4.2.2.2). FlexRay, which is backed by the world's automotive industry, is becoming the standard in the industry and is already used in the BMW X5 model since 2006 [25]. In the following, we choose not to discuss further TTP/C, which, to the best of our knowledge, is no more considered for vehicles but is now used in aircraft electronic systems. However, the important experience gained over the years with TTP/C, in particular regarding fault-tolerance features [36] and their formal validation (Chapter 15), will certainly be beneficial to FlexRay.

4.2.2.1 FlexRay Protocol

A consortium of major companies from the automotive field is currently developing the FlexRay protocol. The core members are BMW, Bosch, Daimler, General Motors, NXP Semiconductors, Freescale Semiconductor, and Volkswagen. The first publicly available specification of the FlexRay protocol has been released in 2004, the current version of the specification [24] is available at http://www.flexray.com.

The FlexRay network is very flexible with regard to topology and transmission support redundancy. It can be configured as a bus, a star, or a multistar. It is not mandatory that each station possesses replicated channels nor a bus guardian, even though this should be the case for critical functions such as the Steer-by-Wire. At the MAC level, FlexRay defines a communication cycle as the concatenation of a TT (or static) window and an event-triggered (or dynamic) window. In each communication window, size of which is set statically at design time, two distinct protocols are applied. The communication cycles are executed periodically. The TT window uses a TDMA MAC protocol; the main difference with TTP/C is that a station in FlexRay might possess several slots in the TT window, but the size of all the slots is identical (Figure 4.1). In the event-triggered part of the communication cycle, the protocol is flexible TDMA (FTDMA): the time is divided into so-called minislots, each station possesses a given number of minislots (not necessarily consecutive) and it can start the transmission of a frame inside each of its own minislots. A minislot remains idle if the station has nothing to transmit, which actually induces a loss of bandwidth (see Ref. [37] for a discussion on this topic). An example of a dynamic window is shown in Figure 4.2: on channel B, frames have been transmitted in minislots n and $n + 2$ while minislot $n+1$ has not been used. It is noteworthy that frame $n+4$ is not received simultaneously on channels A and B since, in the dynamic window, transmissions are independent in both channels.

The FlexRay MAC protocol is more flexible than the TTP/C MAC since in the static window nodes are assigned as many slots as necessary (up to 2047 overall) and since in the dynamic part of the communication cycle frames are only transmitted if necessary. In a similar way as with TTP/C, the structure of the communication cycle is

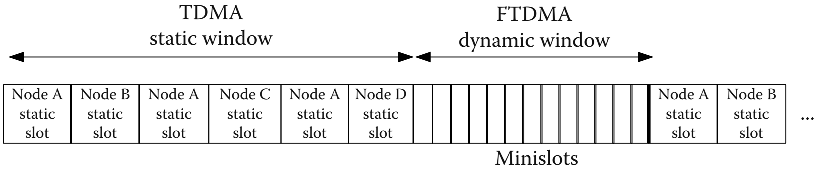


FIGURE 4.1 Example of a FlexRay communication cycle with four nodes A, B, C, and D.

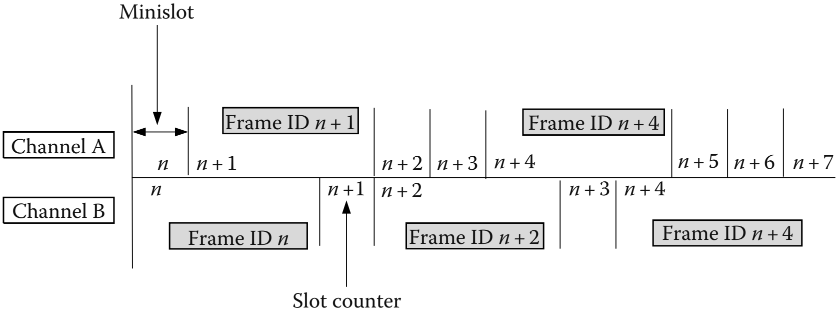


FIGURE 4.2 Example of message scheduling in the dynamic segment of the FlexRay communication cycle.

statically stored in the nodes; however, unlike TTP/C, mode changes with a different communication schedule for each mode are not possible.

The FlexRay frame consists of three parts: the header, the payload segment containing up to 254 bytes of data, and the CRC of 24 bits. The header of 5 bytes includes the identifier of the frame and the length of the data payload. The use of identifiers allows to move a software component, which sends a frame *X*, from one ECU to another ECU without changing anything in the nodes that consume frame *X*. It has to be noted that this is no more possible when signals produced by distinct components are packed into the same frame for the purpose of saving bandwidth (i.e., which is referred to as frame-packing or protocol data unit [PDU]-multiplexing—see Ref. [38] for this problem addressed on CAN).

From the dependability point of view, the FlexRay standard specifies solely the bus guardian and the clock synchronization algorithms. Other features, such as mode management facilities or a membership service, will have to be implemented in software or hardware layers on top of FlexRay (see, for instance, Ref. [39] for a membership service protocol that could be used along with FlexRay). This will allow to conceive and implement exactly the services that are needed with the drawback that correct and efficient implementations might be more difficult to achieve in a layer above the communication controller.

In the FlexRay specification, it is argued that the protocol provides scalable dependability, that is, the "ability to operate in configurations that provide various degrees of fault tolerance." Indeed, the protocol allows for mixing links with single and dual transmission supports on the same network, subnetworks of nodes without bus-guardians, or with different fault-tolerance capability with regards to clock synchronization, etc. In the automotive context where critical and noncritical functions will increasingly coexist and interoperate, this flexibility can prove to be efficient in terms of cost and reuse of existing components if missing fault-tolerance features are provided in a middleware (MW) layer, for instance such as the one currently under development within the automotive industry project AUTOSAR (Section 4.3.3). The reader interested in more information about FlexRay can refer to Chapter 5, and to Refs. [40,41] for how to configure the communication cycle.

4.2.2.2 TTCAN Protocol

TTCAN [27] is a communication protocol developed by Robert Bosch GmbH on top of the CAN physical and DLL. TTCAN uses the CAN standard but, in addition, requires that the controllers have the possibility to disable automatic retransmission of frames upon transmission errors and to provide the upper layers with the point in time at which the first bit of a frame was sent or received [42]. The bus topology of the network, the characteristics of the transmission support, the frame format, as well as the maximum data rate, 1 Mbps, are imposed by the CAN protocol. Channel redundancy is possible (see Ref. [43] for a proposal), but not standardized and no bus guardian is implemented in the node. The key idea is to propose, as with FlexRay, a flexible TT/event-triggered protocol. As illustrated in Figure 4.3, TTCAN defines a basic cycle (the equivalent of the FlexRay communication cycle) as the concatenation of one or several TT (or exclusive) windows and one event-triggered (or arbitrating) window. Exclusive windows are devoted to TT transmissions (i.e., periodic messages) while the arbitrating window is ruled by the standard CAN protocol: transmissions are dynamic and bus access is granted according to the priority of the frames. Several basic cycles that differ by their organization in exclusive and arbitrating windows and by the messages sent inside exclusive windows can be defined. The list of successive basic cycles is called the system matrix, which is executed in loops. Interestingly, the protocol enables the master node (i.e., the node that initiates the basic cycle through the transmission of the "reference message") to stop functioning in TTCAN mode and to resume in standard CAN mode. Later, the master node can switch back to TTCAN mode by sending a reference message.

TTCAN is built on a well-mastered and low-cost technology, CAN, but, as defined by the standard, does not provide important dependability services such as the bus guardian, membership service, and reliable acknowledgment. It is, of course, possible to implement some of these mechanisms at the application or MW level but with reduced efficiency. A few years back, it was thought that carmakers could be interested in using TTCAN during a transition period until FlexRay technology matured fully

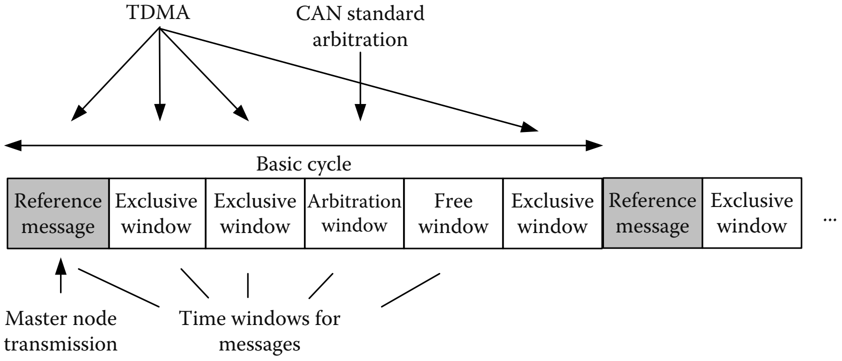


FIGURE 4.3 Example of a TTCAN basic cycle.

but this was not really the case and it seems that the future of TTCAN in production cars is rather unsure.

4.2.3 Low-Cost Automotive Networks

Several fieldbus networks have been developed to fulfill the need for low-speed/low-cost communication inside mechatronic-based subsystems generally made of an ECU and its set of sensors and actuators. Two representatives of such networks are LIN and TTP/A. The low-cost objective is achieved not only because of the simplicity of the communication controllers but also because the requirements set on the microcontrollers driving the communication are reduced (i.e., low computational power, small amount of memory, low-cost oscillator). Typical applications involving these networks include controlling doors (e.g., door locks, opening/closing windows) or controlling seats (e.g., seat position motors, occupancy control). Besides cost considerations, a hierarchical communication architecture, including a backbone such as CAN and several subnetworks such as LIN, enables reducing the total traffic load on the backbone.

Both LIN and TTP/A are master/slave networks where a single master node, the only node that has to possess a precise and stable time base, coordinates the communication on the bus: a slave is only allowed to send a message when it is polled. More precisely, the dialog begins with the transmission by the master of a "command frame" that contains the identifier of the message whose transmission is requested. The command frame is then followed by a "data frame" that contains the requested message sent by one of the slaves or by the master itself (i.e., the message can be produced by the master).

4.2.3.1 LIN

LIN [16,17] is a low-cost serial communication system used as SAE class A network, where the needs in terms of communication do not require the implementation of higher-bandwidth multiplexing networks such as CAN. LIN is developed by a set of major companies from the automotive industry (e.g., DaimlerChrysler, Volkswagen, BMW, and Volvo) and is already widely used in production cars.

The LIN specification package (LIN version 2.1 [16]) includes not only the specification of the transmission protocol (physical and DLL) for master–slave communications but also the specification of a diagnostic protocol on top of the DLL. A language for describing the capability of a node (e.g., bit-rates that can be used, characteristics of the frames published and subscribed by the node, etc.) and for describing the whole network is provided (e.g., nodes on the network, table of the transmissions' schedule, etc.). This description language facilitates the automatic generation of the network configuration by software tools.

A LIN cluster consists of one "master" node and several "slave" nodes connected to a common bus. For achieving a low-cost implementation, the physical layer is defined as a single wire with a data rate limited to 20 kbps due to EMI limitations. The master node decides when and which frame shall be transmitted according to the schedule table. The schedule table is a key element in LIN; it contains the list of frames that are

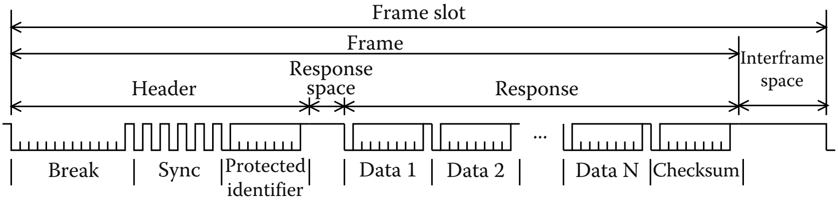


FIGURE 4.4 Format of the LIN frame. A frame is transmitted during its "frame slot," which corresponds to an entry of the schedule table.

to be sent and their associated frame-slots thus ensuring determinism in the transmission order. At the moment a frame is scheduled for transmission, the master sends a header (a kind of transmission request or command frame) inviting a slave node to send its data in response. Any node interested can read a data frame transmitted on the bus. As in CAN, each message has to be identified: 64 distinct message identifiers are available. Figure 4.4 depicts the LIN frame format and the time period, termed a "frame slot," during which a frame is transmitted.

The header of the frame that contains an identifier is broadcast by the master node and the slave node that possesses this identifier inserts the data in the response field. The "break" symbol is used to signal the beginning of a frame. It contains at least 13 dominant bits (logical value 0) followed by one recessive bit (logical value 1) as a break delimiter. The rest of the frame is made of byte fields delimited by one start bit (value 0) and one stop bit (value 1), thus resulting in a 10-bit stream per byte. The "sync" byte has a fixed value (which corresponds to a bit stream of alternatively 0 and 1); it allows slave nodes to detect the beginning of a new frame and be synchronized at the start of the identifier field. The so-called protected identifier is composed of two subfields: the first 6 bits are used to encode the identifier and the last 2 bits, the identifier parity. The data field can contain up to 8 bytes of data. A checksum is calculated over the protected identifier and the data field. Parity bits and checksum enable the receiver of a frame to detect bits that have been inverted during transmission.

LIN defines five different frame types: unconditional, event-triggered, sporadic, diagnostic, and user-defined. Frames of the latter type are assigned a specific identifier value and are intended to be used in an application-specific way that is not described in the specification. The first three types of frames are used to convey signals. Unconditional frames are the usual type of frames used in the master–slave dialog and are always sent in their frame-slots. Sporadic frames are frames sent by the master, only if at least one signal composing the frame has been updated. Usually, multiple sporadic frames are assigned to the same frame-slot and the higher priority frame that has an updated signal is transmitted. An event-triggered frame is used by the master willing to obtain a list of several signals from different nodes. A slave will only answer the master if the signals it produces have been updated, thus resulting in bandwidth savings if updates do not take place very often. If more than one slave answers, a collision will occur. The master resolves the collision by requesting all signals in the list one by one. A typical example of the use of the event-triggered transfer given in Ref. [44]

is the doors' knob monitoring in a central locking system. As it is rare that multiple passengers simultaneously press a knob, instead of polling each of the four doors, a single event-triggered frame can be used. Of course, in the rare event when more than one slave responds, a collision will occur. The master will then resolve the collision by sending one by one the individual identifiers of the list during the successive frame slots reserved for polling the list. Finally, diagnostic frames have a fixed size of 8 bytes, fixed value identifiers for both the master's request and the slave's answers, and always contain diagnostic or configuration data whose interpretation is defined in the specification.

It is also worth noting that LIN offers services to send nodes into a sleep mode (through a special diagnostic frame termed "go-to-sleep-command") and to wake them up, which is convenient since optimizing energy consumption, especially when the engine is not running, is a real matter of concern in the automotive context.

4.2.3.2 TTP/A Network

As with TTP/C, TTP/A [18] was initially invented at the Vienna University of Technology. TTP/A pursues the same aims and shares the main design principles as LIN and it offers, at the communication controller level, some similar functionalities, in particular, in the areas of plug-and-play capabilities and online diagnostics services. TTP/A implements the classic master–slave dialog, termed "master–slave round," where the slave answers the master's request with a data frame having a fixed length data payload of 4 bytes. The "multipartner" rounds enable several slaves to send up to an overall amount of 62 bytes of data after a single command frame. A "broadcast round" is a special master–slave round in which the slaves do not send data; it is, for instance, used to implement sleep/wake-up services. The data rate on a single wire transmission support is, as for LIN, equal to 20 kbps, but other transmission supports enabling higher data rates are possible. To the best of our knowledge, TTP/A is not currently in use in production cars.

4.2.4 Multimedia Networks

Many protocols have been adapted or specifically conceived for transmitting the large amount of data needed by emerging multimedia applications in automotive systems. Two prominent protocols in this category are MOST and IDB-1394.

4.2.4.1 MOST Network

MOST [22] is a multimedia network development which was initiated in 1998 by the MOST Cooperation (a consortium of carmakers and component suppliers). MOST provides point-to-point audio and video data transfer with different possible data rates. This supports end-user applications like radios, global positioning system (GPS) navigation, video displays, and entertainment systems. MOST's physical layer is a plastic optical fiber (POF) transmission support that provides a much better resilience to EMI and higher transmission rates than classical copper wires. Current production cars from BMW and DaimlerChrysler employ a MOST network, and MOST has now become the de facto standard for transporting audio and video within vehicles [45,46].

At the time of writing, the third revision of MOST has been announced with, as a new feature, the support of a channel that can transport standard Ethernet frames.

4.2.4.2 IDB-1394 Network

IDB-1394 is an automotive version of IEEE-1394 for in-vehicle multimedia and telematic applications jointly developed by the IDB Forum (see http://www.idbforum.org) and the 1394 Trade Association (see http://www.1394ta.org). The system architecture of IDB-1394 permits existing IEEE-1394 consumer electronics devices to interoperate with embedded automotive grade devices. IDB-1394 supports a data rate of 100 Mbps over twisted pair or POF, with a maximum number of embedded devices that are limited to 63 nodes. From the point of view of transmission rate and interoperability with existing IEEE-1394 consumer electronic devices, IDB-1394 was at one time considered a serious competitor for MOST technology but, despite a few early implementations at Renault and Nissan, did not receive wide acceptance on the market.

4.3 Middleware Layer

4.3.1 Rationale for a Middleware

The design of automotive electronic systems has to take into account several constraints. First, the performance, quality, and safety of a vehicle depend on functions that are mainly implemented in software and moreover depend on a tight cooperation between these functions (Chapter 1). Second, in-vehicle embedded systems are produced through a complex cooperative multipartner development process shared between original equipment manufactures (OEMs) and suppliers. In order to increase the efficiency of the production of components and their integration, two important problems have to be solved: (1) the portability of components from one ECU to another enabling some flexibility in the architecture design, and (2) the reuse of components between platforms, which is a key point especially for ECU suppliers. So the cooperative development process raises the problem of interoperability of components. A classic approach for easing the integration of software components is to implement a Middleware layer that provides application programs with common services and a common interface. In particular, the common interface allows the design of an application disregarding the hardware platform and the distribution, and therefore enables the designer focusing on the development and the validation of the software components and the software architecture that realize a function.

Among the set of common services usually provided by a MW, those that related to the communication between several application components are crucial. They have to meet several objectives:

- *Hide the distribution* through the availability of services and interfaces that are the same for intra-ECU, inter-ECU, interdomain communications whatever the underlying protocols.
- *Hide the heterogeneity* of the platform (i.e., microcontrollers, protocols, operating systems, etc.) by providing an interface independent of the

underlying protocols and of the CPU architecture (e.g., 8/16/32 bits, endianness).

- *Provide high-level services* in order to shorten the development time and increase quality through the reuse of validated services (e.g., working mode management, redundancy management, membership service, etc.). A good example of such a function is the "frame-packing" (sometimes also called "signal multiplexing") that enables application components to exchange "signals" (e.g., the number of revolutions per minute, the speed of the vehicle, the state of a light, etc.) while, at runtime, "frames" are transmitted over the network; so, the frame-packing service of a MW consists in packing the signals into frames and sending the frames at the right points in time for ensuring the deadline constraint on each signal it contains.
- *Ensure QoS properties required by the application*; in particular, it can be necessary to improve the QoS provided by the lower-level protocols as, for example, by furnishing an additional CRC, transparent to the application, if the Hamming distance of the CRC specified by the network protocol is not sufficient with regard to the dependability objectives. Other examples are the correction of "bugs" in lower level protocols such as the "inconsistent message duplicate" of CAN (see Chapter 6 and Ref. [47]), the provision of a reliable acknowledgment service on CAN, the status information on the data consumed by the application components (e.g., data were refreshed since last reading, its freshness constraint was not respected, etc.), or filtering mechanisms (e.g., notify the application for each *k* reception or when the data value has changed in a significant way).

Note that a more advanced feature would be to come up with adaptive communication services, thanks to algorithms that would modify at runtime the parameters of the communication protocols (e.g., priorities, transmission frequencies, etc.) according to the current requirements of the application (e.g., inner-city driving or highway driving) or changing environmental conditions (e.g., EMI level). For the time being, to the best of our knowledge, no such feature exists in automotive-embedded systems. In fact, this point requires a coordinated approach for the design of function (as the definition of control law parameters, the identification of the parameters acting on the robustness of the function, etc.) and the deployment of the software architecture that implements the function (specifically the communication parameters). By increasing the efficiency and the robustness of the application, such an adaptive strategy would certainly ease the reusability.

4.3.2 Automotive MWs Prior to AUTOSAR

Some carmakers possess a proprietary MW that helps to integrate ECUs and software modules developed by their third-party suppliers. For instance, the TITUS/DBKOM communication stack is a proprietary MW of Daimler that standardizes the cooperation between components according to a client/server model. Volcano [48–50] is a commercial product of Mentor Graphics initially developed in partnership with Volvo. The Volcano target package (VTP) consists of a communication layer and a set of off-line configuration tools for application distributed on CAN and/or LIN. It is aimed to provide the mapping of signals into frames under network bandwidth optimization and ensure a predictable and deterministic real-time communication system thanks to schedulability analysis techniques [31,48]. To the best of our knowledge, no publicly available technically precise description of TITUS and Volcano exists.

The objective of the OSEK/VDX consortium (offene Systeme und deren schnittstellen für die Elektronik im Kraftfahrzeug, see http://www.osek-vdx.org) is to build a standard architecture for in-vehicle control units. Among the results of this group, two specifications are of particular interest in the context of this chapter: the OSEK/VDX communication layer [51] and the fault-tolerant communication layer [52]. The OSEK/VDX consortium (http://www.osek-vdx.org) specifies a communication layer [51] that defines common software interfaces and common behavior for internal and external communications between application components. At the application layer, these components exchange signals, termed "messages" in OSEK/VDX terminology, while communicating OSEK/VDX entities exchange so-called interaction layer PDUs (I-PDUs) that are collections of messages. Each consumer of a message can specify it as queued or unqueued (i.e., a new value overwrites the old one) and associate it with a filtering mechanism. The emission of an I-PDU onto the network can be specified as triggered by the sending of a message that it contains or not. In the latter case, the emission of the I-PDU is asynchronous with the sending of the message. How signals are packed into a frame is statically defined off-line and the OSEK/VDX communication layer automatically realizes the packing/unpacking at runtime. The characteristic of I-PDU and messages are specified through the OSEK/VDX implementation language [53].

OSEK/VDX communication runs on top of a transport layer [54] that takes care mainly of the I-PDU segmentation and it can operate on any OS compliant with OSEK/VDX OS services for tasks, events, and interrupt management [55]. Some questions deserve to be raised. In particular, communications between application processes that are internal to one ECU or located in two distant ECUs do not obey exactly the same rules (see Ref. [56] for more details); thus, the designer has to take into account the distribution of the functions that is a hindrance to portability. Finally, OSEK/VDX communication does not follow a TT approach and is not intended to be used on top of a TT network, as for example TTP/C or FlexRay. These networks implement some features that were specified in OSEK/VDX communication, as the TT sending of I-PDU, while some that are offered by this MW are not compatible with the TT paradigm, as the direct transmission of an I-PDU as soon as a message that it contains is sent by the application. However, higher-level services are still needed on top of FlexRay or TTP/C for facilitating the development of fault-tolerant applications. OSEK/VDX FTCom (fault-tolerant communication) [52] is a proposal whose objective is to complete OSEK/VDX for TT distributed architectures. One of its main functions is to manage the redundancy of data needed for achieving fault-tolerance (i.e., the same information can be produced by a set of replicated nodes) by presenting only one copy of data to the receiver application according to the agreement strategy specified by the designer. Two other important services of the FTCom, that are also provided by OSEK communication are (1) to manage the packing/unpacking of messages [38], which is needed if the network bandwidth has to be optimized (Section 4.4.1), and (2) to provide message filtering mechanisms for passing only "significant" data to the application. OSEK/VDX FTCom was developed to run on top of a TT operating system (OS) such as OSEK time [57]. In this OS, the scheduling of tasks is specified in a timetable called the dispatcher table that is generated off-line. OSEK/VDX FTCom allows the OS to synchronize the start of the task schedule defined in the dispatcher table to a particular point in time in the I-PDU schedule (i.e., the TDMA round). As this point is shared by all the ECUs connected on the same network, this service can be used to synchronize distant applications.

Between 2001 and 2004, a European cooperative project aimed at the specification of an automotive MW within the automotive industry was undertaken (ITEA EAST-EEA project, see http://www.east-eea.net). To the best of our knowledge, the ITEA EAST-EEA project was the first important initiative targeting the specification of both the services to be ensured by the MW and the architecture of the MW itself in terms of components and architecture of components. Similar objectives guide the work done in the AUTOSAR consortium, see Chapter 2 and Refs. [58,59], that gathers most of the key players in the automotive industry. The specifications produced by the consortium become quickly de facto standards for the cooperative development of in-vehicle embedded systems (see, for instance, the migration to AUTOSAR at PSA Peugeot-Citröen [60]).

4.3.3 AUTOSAR

AUTOSAR (AUTomotive Open Standard ARchitecture) specifies the software architecture embedded in an ECU. More precisely, it provides a reference model that is comprised of three main parts:

- Application layer
- Basic software (MW software components)
- Runtime environment (RTE) that provides standardized software interfaces to the application software

One of AUTOSAR's main objectives is to improve the quality and the reliability of embedded systems. By using a well-suited abstraction, the reference model supports the separation between software and hardware and eases the mastering of the complexity. It also allows the portability of application software components and therefore the flexibility for product modification, upgrade, and update, as well as the scalability of solutions within and across product lines. The AUTOSAR reference architecture is schematically illustrated in Figure 4.5. An application software component is compliant with AUTOSAR if its code only calls entry points defined by the RTE. Furthermore, a basic software component used within the MW has to be of one of the types defined in AUTOSAR; it is AUTOSAR compliant if it provides the services and the interface formally defined in the specification of its type. The generation of an AUTOSAR MW is done from the basic software components, generally provided

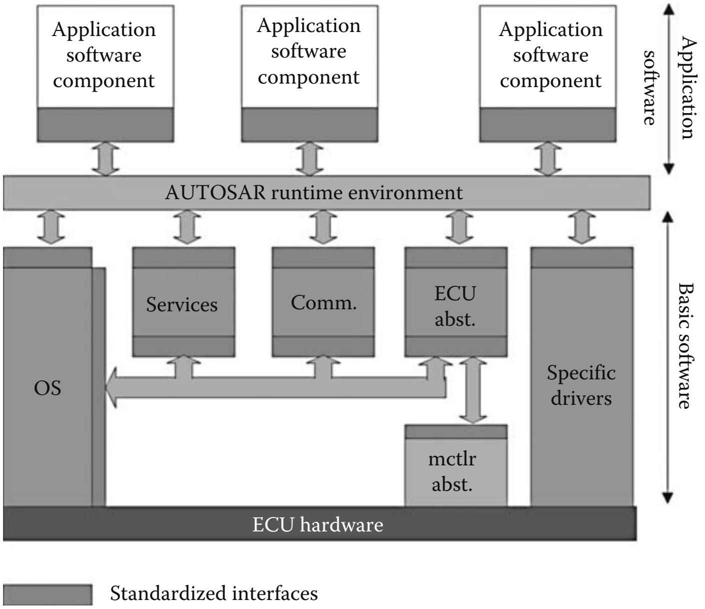


FIGURE 4.5 AUTOSAR reference architecture.

by suppliers, and the specification of the application itself (description of applicative-level tasks, signals sent or received, events, alarms, etc.). Therefore, its deployment can be optimized for each ECU.

One of the main objectives of the AUTOSAR MW is to hide the characteristic of the hardware platform as well as the distribution of the application software components. Thus the inter- or intra-ECU communication services are of major importance and are thoroughly described in the documents provided by the AUTOSAR consortium (see Figure 4.6 for an overview of the different modules). The role of these services is crucial for the behavioral and temporal properties of an embedded and distributed application. Thus their design and configuration have to be precisely mastered and the verification of timing properties becomes an important activity. The problem is complex because the objects (e.g., signals, frames, I-PDU, etc.) that are handled by services at one level are not the same objects that are handled by services at another level. Nevertheless, each object is strongly dependent on one or several objects handled by services belonging to neighboring levels. The AUTOSAR standard proposes two communication models:

- "Sender/receiver" used for passing information between two application software components (belonging to the same task, to two distinct tasks, on the same ECU or to two remote tasks).
- "Client/server" that supports function invocation.

Two communication modes are supported for the sender/receiver communication model:

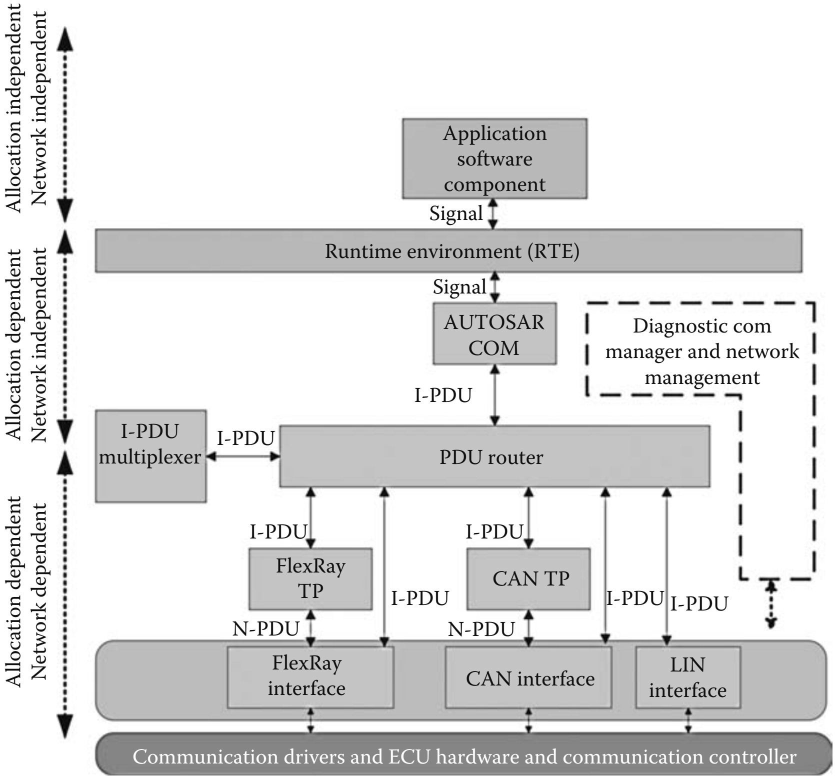


FIGURE 4.6 Communication software components and architecture.

- The "explicit" mode is specified by a component that makes explicit calls to the AUTOSAR MW for sending or receiving data.
- The "implicit" mode means that the reading (resp. writing) of data is automatically done by the MW before the invocation (resp. after the end of execution) of a component consuming (resp. producing) the data without any explicit call to AUTOSAR services.

AUTOSAR identifies three main objects regarding the communication: signal exchanged between software components at application level, I-PDU that consists of a group of one or several signals, and the N-PDU (DLL PDU) that will actually be transmitted on the network. Precisely AUTOSAR defines

- *Signals* at application level that are specified by a length and a type. Conceptually a signal is exchanged between application software components through ports disregarding the distribution of this component. The application needs to precise a transfer property parameter that will impact the behavior of the transmission:
  - The value "triggered" for this parameter indicates that each time the signal is provided to the MW by the application, it has to be transmitted on the network (as we will see later, this means that the sending of the

frame containing this signal is directly done after the emission of the signal by the application component).

  - On the contrary, the value "pending" for a signal indicates that its actual transmission on the network depends only on the emission rule of the frame that contains the signal.

  Furthermore, when specifying a signal, the designer has to indicate if it is a "data" or an "event." In the former case, incoming data are not queued on the receiver side: when a new value arrives, it erases the previous value of the same signal. The latter case specifies that signals are queued on the receiver side and therefore it ensures that for each transmission of the signal a new value will be made available to the application. The handling of buffers or queues is done by the RTE.

- I-PDU are built by the AUTOSAR COM component. Each I-PDU is made of one or several signals and is passed via the PDU router to the communication interfaces. The maximum length of an I-PDU depends on the maximum length of the L-PDU (i.e., DLL PDU) of the underlying communication interface: for CAN and LIN the maximum L-PDU length is 8 bytes while for FlexRay the maximum L-PDU length is 254 bytes. AUTOSAR COM ensures a local transmission when both components are located on the same ECU, or by building suited objects and triggering the appropriate services of the lower layers when the components are remote. This scheme enables the portability of components and hides their distribution. The transformation from signals to I-PDU and from I-PDU to signals is done according to an off-line generated configuration. Each I-PDU is characterized by a behavioral parameter, termed transmission mode with different possible values:
  - "Direct" indicates that the sending of the I-PDU is done as soon as a "triggered" signal contained in this I-PDU is sent at application layer.
  - "Periodic" means that the sending of the I-PDU is done only periodically; it imposes that the I-PDU does not contain triggered signals.
  - "Mixed" means that the rules imposed by the triggered signals contained in the I-PDU are taken into account, and additionally the I-PDU is sent periodically if it contains at least one "pending" signal.
  - "None" characterizes I-PDUs whose emission rules depend on the underlying network protocol (e.g., FlexRay) and no transmission is initiated by AUTOSAR COM in this mode.
- An N-PDU is built by the basic components CAN TP (Transport Protocol) or FlexRay TP. It consists of the data payload of the frame that will be transmitted on the network and protocol control information. Note that the use of a transport layer is not mandatory and I-PDUs can be transmitted directly to the lower layers (Figure 4.6). When a transport layer is used, an N-PDU is obtained by:
  - Splitting the I-PDU so as to obtain several N-PDUs that are compliant with the frame data payload length
  - Assembling several I-PDUs into one N-PDU

The RTE implements the AUTOSAR MW interface and the corresponding services. In particular, the RTE handles the “implicit/explicit” communication modes and the fact that the communication involves events (queued) or data (unqueued). Figure 4.7 illustrates how the transmission of a signal S between two remote application components (ASC-S on the sender side and ASC-R on the receiver side) is handled by the RTE and the COM components. Signal S is assumed to be a data; therefore it is not queued, and it is received explicitly (explicit mode). On each ECU, the RTE is generated according to the specification of the signal exchanges between applicative-level components. Thus, in particular, on the receiver side, a buffer is defined in the RTE for each data that is received by this ECU. At the initialization of the system, the value of signal S at the receiver end is set to a statically defined value (in the example of Figure 4.7, the initial value is 0). The buffer contains a value 0 between t_1 and t_3, and a value 20 from time t_3 on. The value returned by a read call done by the application software component ASC-R on the receiver side is 0 thus at time t_2 and 20 at time t_4.

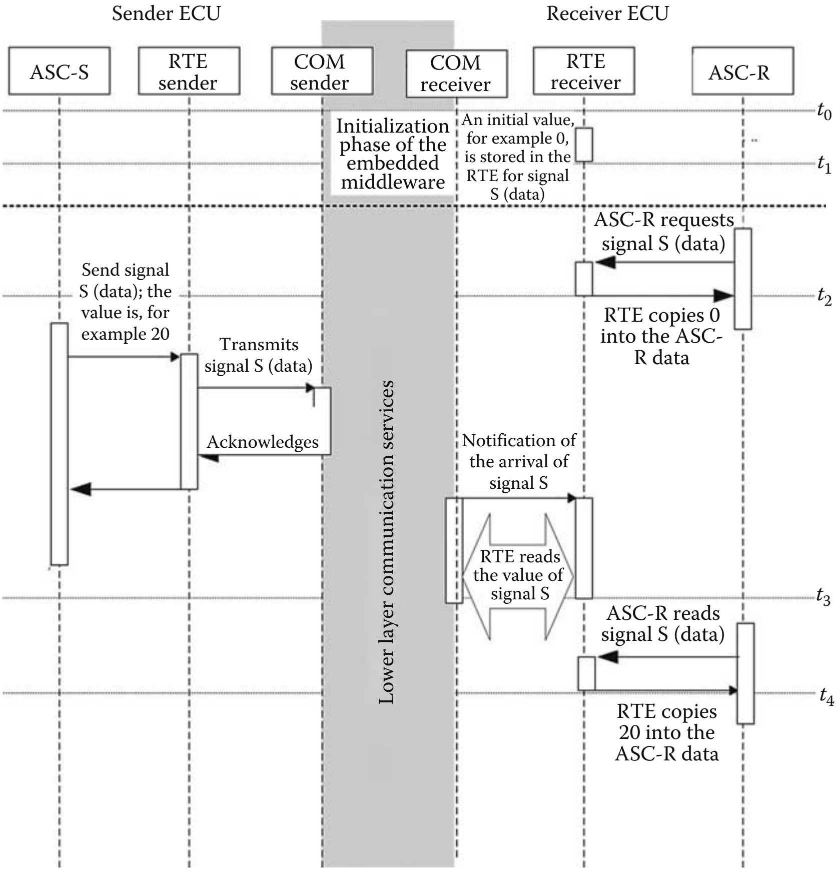

Sender ECU
Receiver ECU
ASC-S
RTE sender
COM sender
COM receiver
RTE receiver
ASC-R
t_0
t_1
t_2
t_3
t_4
Initialization phase of the embedded middleware
An initial value, for example 0, is stored in the RTE for signal S (data)
ASC-R requests signal S (data)
RTE copies 0 into the ASC-R data
Send signal S (data); the value is, for example 20
Transmits signal S (data)
Acknowledges
Lower layer communication services
Notification of the arrival of signal S
RTE reads the value of signal S
ASC-R reads signal S (data)
RTE copies 20 into the ASC-R data

FIGURE 4.7 Sender/receiver communication model for a data signal according to the explicit communication mode.

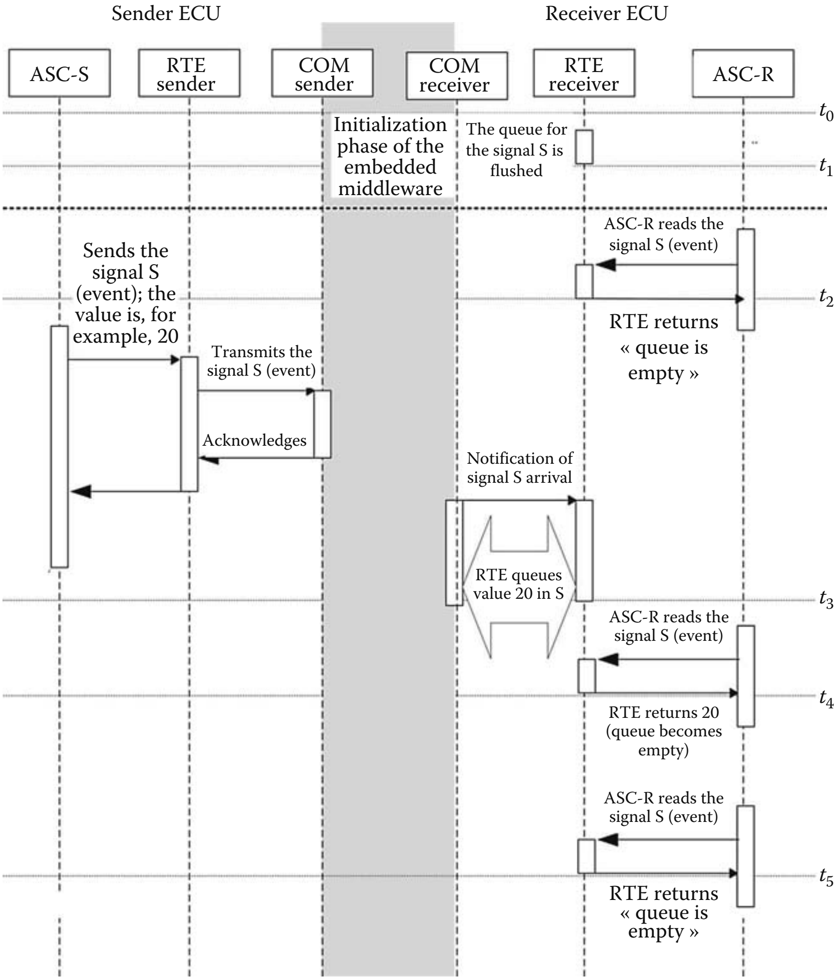

Sender ECU
Receiver ECU
ASC-S
RTE sender
COM sender
COM receiver
RTE receiver
ASC-R
t_0
Initialization phase of the embedded middleware
The queue for the signal S is flushed
t_1
ASC-R reads the signal S (event)
Sends the signal S (event); the value is, for example, 20
t_2
RTE returns « queue is empty »
Transmits the signal S (event)
Acknowledges
Notification of signal S arrival
RTE queues value 20 in S
t_3
ASC-R reads the signal S (event)
t_4
RTE returns 20 (queue becomes empty)
ASC-R reads the signal S (event)
t_5
RTE returns « queue is empty »

FIGURE 4.8 Sender/receiver communication model for an event signal according to the explicit communication mode.

In Figure 4.8, a similar example is given but this time signal S is an "event" and thus it is queued by the RTE on the receiving ECU. At time t_1, the queue for S is initialized (queue is empty). Value 20 is queued at time t_3, at t_4 a read call done by the receiver application component returns 20 and the queue becomes empty. At times t_2 and t_5 such a read call returns a code indicating that the queue is empty.

The AUTOSAR COM component is responsible for several functions: on the sender side, it ensures the transmission and notifies the application about its outcome (success or error). In particular, AUTOSAR COM can inform the application if the transmission of an I-PDU did not take place before a specified deadline (i.e., deadline monitoring). On the receiver side, it also notifies the application (success or error of a reception) and supports the filtering mechanism for signals (dispatching each signal of a received I-PDU to the application or to a gateway). Both at the sending and receiving end, the endianness conversion is taken in charge. An important role of the COM component is to pack/unpack signals into/from I-PDUs. Note that as the maximal length of an I-PDU depends on the underlying networks, the design of a COM component has to take into account the networks and therefore it is not fully independent of the hardware architecture. The COM component has also to determine the points in time when to send the I-PDUs. This is based on the attributes transmission mode of an I-PDU and on the attribute transfer property of each signal that it contains. Figure 4.9 summarizes the combinations that are possible. Note that the "none" transmission mode is not indicated in this table, in that case the transmission is driven by the underlying network layers.

As can be seen in Figure 4.9, the actual sending of an I-PDU and therefore of the signals that it contains is relevant to several rules. Figure 4.10 illustrates how a direct I-PDU containing two signals S_1 and S_2 triggered is transmitted. At times t_1, t_2, t_3, t_4, the sending of signal S_1 or of signal S_2 to the COM component triggers the emission of the I-PDU to the lower layer. In Figure 4.11, we consider an I-PDU in which are packed signal S_1 (triggered) and signal S_2 (pending). The transmission mode of the I-PDU is set to mixed with period dt. Each time a new value of S_1 is provided to the RTE, the I-PDU is passed to the lower layer (times t_1 and t_6). In addition, the I-PDU is also transmitted every dt (times t_4 and t_5). Note that, in this configuration, some values of S_2 may not be transmitted, as for example the value of S_2 provided at time t_2.

Transfer property of the signals	All the signals in the I-PDU are triggered	All the signals in the I-PDU are pending	At least one signal is triggered and one is pending in the I-PDU
Transmission mode of the I-PDU Direct	The transmission of the I-PDU is done each time a signal is sent		This configuaration could be dangerous: if no emission of triggered signals occurs, the pending signals will never be transmitted
Periodic	The transmission of the I-PDU is done periodically	The transmission of the I-PDU is done periodically	The transmission of the I-PDU is done periodically
Mixed	The transmission of the I-PDU is done each time a signal is sent and at each period	The transmission of the I-PDU is done periodically	The transmission of the I-PDU is done each time a triggered signal is sent and at each period

FIGURE 4.9 Transmission mode of an I-PDU versus transfer property of its signals.

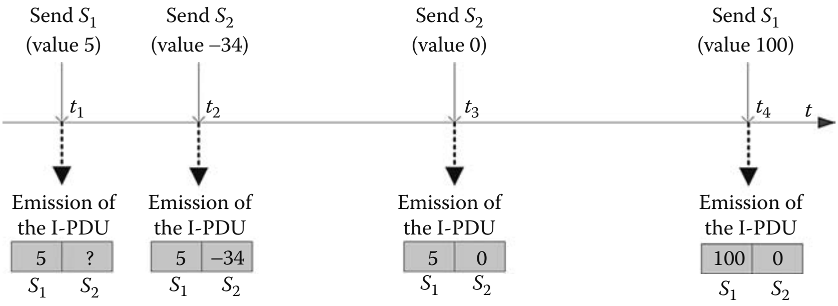


FIGURE 4.10 Transmission of an I-PDU in direct mode with two triggered signals.

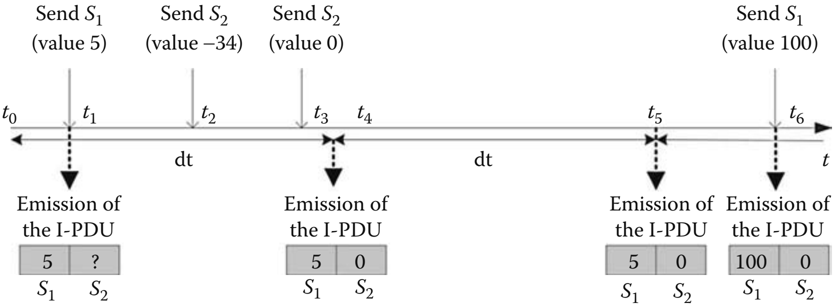


FIGURE 4.11 Transmission of an I-PDU in mixed mode that contains a triggered signal (S_1) and a pending signal (S_2).

The COM component is generated off-line on the basis of the knowledge of the signals, the I-PDUs, and the allocation of application software components on the ECUs. The AUTOSAR PDU router (Figure 4.6), according to the configuration, dispatches each I-PDU to the right network communication stack. This basic component is statically generated off-line as soon as the allocation of software components and the operational architecture is known. Other basic software components of the communication stack are responsible for the segmenting/reassembling of I-PDU(s) when needed (FlexRay TP, CAN TP) or for providing an interface to the communication drivers (FlexRay interface, CAN interface, LIN interface).

4.4 Open Issues for Automotive Communication Systems

4.4.1 Optimized Networking Architectures

The traditional partitioning of the automotive application into several distinct functional domains with their own characteristics and requirements is useful in mastering the complexity, but this leads to the development of several independent subsystems with their specific architectures, networks, and software technologies.

Some difficulties arise from this partitioning since more and more cross-domain data exchanges are needed. This requires implementing gateways whose performances in terms of CPU load and impact on data freshness have to be carefully assessed (see, for instance, Ref. [61]). For instance, an ECU belonging, from a functional point of view, to a particular domain can be connected, for wiring reasons, onto a network of another domain. For example, the diesel particulate filter (DPF) is connected onto the body network in some vehicles even though it belongs, from a functional standpoint, to the power train. This can raise performance problems since the DPF needs a stream of data with strong temporal constraints coming from the engine controller located on the power train network. Numerous other examples of cross-domain data exchanges can be cited such as the engine controller (power train) that takes input from the climate control (body) or information from the power train displayed on the dashboard (body). There are also some functions that one can consider as being cross-domains such as the immobilizer, which belongs both to the body and power train domains. Upcoming X-by-Wire functions will also need very tight cooperation between the ECUs of the chassis, the power train, and the body.

A current practice is to transfer data between different domains through a gateway usually called the "central body electronic," belonging to the body domain. This subsystem is recognized as being critical in the vehicle: it constitutes a single point of failure, its design is overly complex, and performance problems arise due to an increasing workload.

An initial foreseeable domain of improvement is to further develop the technologies needed for the interoperability between applications located on different subnetworks. With the AUTOSAR project, significant progresses in the area of MW have been achieved over the last years and we are coming closer to the desirable characteristics listed in Section 4.3.1.

Future work should also be devoted to optimizing networking architectures. This implies rethinking the current practice that consists of implementing networks on a per-domain basis. The use of technologies that could fulfill several communication requirements (e.g., high-speed, event-triggered, and TT communication, all possible with FlexRay) with scalable performances is certainly one possible direction for facilitating the design. Certainly, software tools, such as our tool NETCAR-Analyzer (see http://www.realtimeatwork.com), will be helpful to master the complexity and come up with cost and dependability-optimized solutions. The use of software along the development cycle will be facilitated by the advent of the ASAM FIBEX standard [62], in the process of being adopted by AUTOSAR, which enables to fully describe the networks embedded in a vehicle (CAN, LIN, FlexRay, MOST, and TTCAN protocols), the frames that are exchanged between ECUs, and the gatewaying strategies.

4.4.2 System Engineering

The verification of the performances of a communication system is twofold. On the one hand, some properties of the communication system services can be proved independently of the application. For instance, the correctness of the synchronization and the membership and clique avoidance services of TTP/C have been studied using formal methods in Refs. [39,63,64].

There are other constraints whose fulfillment cannot be determined without a precise model of the system. This is typically the case for real-time constraints on tasks and signals where the patterns of activations and transmissions have to be identified. Much work has already been done in this field during the last 10 years: schedulability analysis on priority buses [31], joint schedulability analysis of tasks and messages [65,66], probabilistic assessment of the reliability of communications under EMI [34,36,67,68], etc. What is now needed is to extend these analyses to take into account the peculiarities of the platforms in use (e.g., overheads due to the OS and the stack of communication layers) and to integrate them in the development process of the system. The problem is complicated by the development process being shared between several partners (the carmaker and various third-part suppliers). Methods have to be devised to facilitate the integration of components developed independently and to ensure their interoperability.

In terms of the criticality of the involved functions, future automotive X-by-Wire systems can reasonably be compared with Flight-by-Wire systems in the avionic field. According to Ref. [69], the probability of encountering a critical safety failure in vehicles must not exceed 5×10^{-10}/h and per system, but other studies consider 10^{-9}. It will be a real challenge to reach such dependability, in particular, because of the cost constraints. It is certain that the know-how gathered over the years in the avionic industry can be of great help but design methodologies adapted to the automotive constraints have to be developed.

The first step is to develop technologies able to integrate different subsystems inside a domain (Section 4.4.1) but the real challenge is to shift the development process from subsystem integration to a complete integrated design process. The increasing amount of networked control functions inside in-car embedded systems leads to developing specific design processes based, among others, on formal analysis and verification techniques of both dependability properties of the networks and dependability requirements of the embedded application.

References

1. A. Albert. Comparison of event-triggered and time-triggered concepts with regards to distributed control systems. In *Proceedings of Embedded World 2004*, Nuremberg, Germany, February 2004.
2. G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1), January 2002.
3. K. Johansson, M. Törngren, and L. Nielsen. Applications of controller area network. In *Handbook of Networked and Embedded Control Systems*, D. Hristu-Varsakelis and W.S. Levine (Eds.), Birkhuser, Boston, MA, 2005.
4. C. Wilwert, N. Navet, Y.-Q. Song, and F. Simonot-Lion. Design of automotive X-by-wire systems. In *The Industrial Communication Technology Handbook*, R. Zurawski (Ed.), CRC Press, Boca Raton, FL, 2005.

5. M. Ayoubi, T. Demmeler, H. Leffler, and P. Köhn. X-by-Wire functionality, performance and infrastructure. In *Proceedings of Convergence 2004*, Detroit, MI, 2004.
6. M. Krug and A.V. Schedl. New demands for in-vehicle networks. In *Proceedings of the 23rd EUROMICRO Conference'97*, Budapest, Hungary, July 1997.
7. M. Peteratzinger, F. Steiner, and R. Schuermans. Use of XCP on FlexRay at BMW. Translated reprint from HANSER Automotive 9/2006, available at url https://www.vector-worldwide.com/vi_downloadcenter_en, 223.html?product=xcp, 2006.
8. S. Poledna, W. Ettlmayr, and M. Novak. Communication bus for automotive applications. In *Proceedings of the 27th European Solid-State Circuits Conference*, Villach, Austria, September 2001.
9. J. Rushby. A comparison of bus architecture for safety-critical embedded systems. Technical report, NASA/CR, March 2003.
10. K. Ramaswamy and J. Cooper. Delivering multimedia content to automobiles using wireless networks. In *Proceedings of Convergence 2004*, Detroit, MI, 2004.
11. Ford Motor Company. Ford to study in-vehicle electronic devices with advanced simulators. Available at url http://media.ford.com/article_display. cfm?article_id=7010, 2001.
12. A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. In *Proceedings of the 3rd Information Survivability Workshop*, Boston, MA, 2000, pp. 7–12.
13. Intel Corporation. Introduction to in-vehicle networking. Available at http://support.intel.com/design/auto/autolxbk.htm, 2004.
14. Society of Automotive Engineers. J2056/1 class C application requirements classifications. In *SAE Handbook*, 1994.
15. Society of Automotive Engineers. J2056/2 survey of known protocols. In *SAE Handbook*, Vol. 2, 1994.
16. LIN Consortium. *LIN Specification Package, revision 2.1*, November 2006. Available at http://www.lin-subbus.org/.
17. A. Rajnàk. The LIN standard. In *The Industrial Communication Technology Handbook*, R. Zurawski (Ed.), CRC Press, Boca Raton, FL, 2005.
18. H. Kopetz et al. *Specification of the TTP/A Protocol*. University of Technology, Vienna, Austria, September 2002.
19. Society of Automotive Engineers. Class B data communications network interface—SAE J1850 standard—rev. Nov. 96, 1996.
20. International Standard Organization. *ISO 11519-2, Road Vehicles—Low Speed Serial Data Communication—Part 2: Low Speed Controller Area Network*. ISO, 1994.
21. International Standard Organization. *ISO 11519-3, Road Vehicles—Low Speed Serial Data Communication—Part 3: Vehicle Area Network (VAN)*. ISO, 1994.
22. MOST Cooperation. *MOST Specification Revision 2.3*, August 2004. Available at http://www.mostnet.de.
23. TTTech Computertechnik GmbH. *Time-Triggered Protocol TTP/C, High-Level Specification Document, Protocol Version 1.1*, November 2003. Available at http://www.tttech.com.
24. FlexRay Consortium. FlexRay communications system—protocol specification—version 2.1. Available at http://www.flexray.com, December 2005.

25. A. Schedl. Goals and architecture of FlexRay at BMW. In *Slides Presented at the Vector FlexRay Symposium*, March 2007.
26. P. Bühring. Safe-by-Wire Plus: Bus communication for the occupant safety system. In *Proceedings of Convergence 2004*, Detroit, MI, 2004.
27. International Standard Organization. *11898-4, Road Vehicles—Controller Area Network (CAN)—Part 4: Time-Triggered Communication*. ISO, 2000.
28. J. Ferreira, P. Pedreiras, L. Almeida, and J.A. Fonseca. The FTT-CAN protocol for flexibility in safety-critical systems. *IEEE Micro Special Issue on Critical Embedded Automotive Networks*, 22(4):46–55, July–August 2002.
29. P. Koopman. Critical embedded automotive networks. *IEEE Micro Special Issue on Critical Embedded Automotive Networks*, 22(4):14–18, July–August 2002.
30. N. Navet and Y.-Q. Song. Validation of real-time in-vehicle applications. *Computers in Industry*, 46(2):107–122, November 2001.
31. K. Tindell, A. Burns, and A.J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.
32. Y. Martin. L'avenir de l'automobile tient à un fil. *L'argus de l'automobile*, 3969:22–23, March 2005.
33. International Standard Organization. *ISO 11898, Road Vehicles—Interchange of Digital Information—Controller Area Network for High-speed Communication*. ISO, 1994.
34. B. Gaujal and N. Navet. Fault confinement mechanisms on CAN: Analysis and improvements. *IEEE Transactions on Vehicular Technology*, 54(3):1103–1113, May 2005.
35. M. Waern. Evaluation of protocols for automotive systems. Master's thesis, KTH Machine Design, Stockholm, Sweden, 2003.
36. B. Gaujal and N. Navet. Maximizing the robustness of TDMA networks with applications to TTP/C. *Real-Time Systems*, 31(1–3):5–31, December 2005.
37. G. Cena and A. Valenzano. Performance analysis of Byteflight networks. In *Proceedings of the 2004 IEEE Workshop of Factory Communication Systems (WFCS 2004)*, Vienna, Austria, September 2004, pp. 157–166.
38. R. Saket and N. Navet. Frame packing algorithms for automotive applications. *Journal of Embedded Computing*, 2:93–102, 2006.
39. R. Barbosa and J. Karlsson. Formal specification and verification of a protocol for consistent diagnosis in real-time embedded systems. In *Third IEEE International Symposium on Industrial Embedded Systems (SIES'2008)*, Montpellier, France, June 2008.
40. M. Grenier, L. Havet, and N. Navet. Configuring the communication on FlexRay: The case of the static segment. In *ERTS Embedded Real Time Software 2008*, Toulouse, France 2008.
41. T. Pop, P. Pop, P. Eles, and Z. Peng. Bus access optimisation for FlexRay-based distributed embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '07)*, San Jose, CA, 2007, pp. 51–56. EDA Consortium.
42. Robert Bosch GmbH. Time triggered communication on CAN: TTCAN. Available at http://www.semiconductors.bosch.de/en/20/ttcan/index.asp, 2008.
43. B. Müller, T. Führer, F. Hartwich, R. Hugel, and H. Weiler. Fault tolerant TTCAN networks. In *Proceedings of the 8th International CAN Conference (iCC)*, Las Vegas, NV, 2002.

44. LIN Consortium. *LIN Specification Package, version 1.3*, December 2002. Available at http://www.lin-subbus.org/.
45. H. Muyshondt. Consumer and automotive electronics converge: Part 1—Ethernet, USB, and MOST. Available at http://www.automotive-designline.com/, February 2007.
46. H. Muyshondt. Consumer and automotive electronics converge: Part 2—a MOST implementation. Available at http://www.automotivedesignline.com/, March 2007.
47. L.M. Pinho and F. Vasques. Reliable real-time communication in can networks. *IEEE Transactions on Computers*, 52(12):1594–1607, 2003.
48. L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano—a revolution in on-board communications. Technical report, Volvo, 1999.
49. A. Rajnàk. Volcano—enabling correctness by design. In *The Embedded Systems Handbook*, R. Zurawski (Ed.), CRC Press, Boca Raton, FL, 2005.
50. A. Rajnàk and M. Ramnefors. The Volcano communication concept. In *Proceedings of Convergence 2002*, Detroit, MI, 2002.
51. OSEK Consortium. *OSEK/VDX Communication, Version 3.0.3*, July 2004. Available at http://www.osek-vdx.org/.
52. OSEK Consortium. *OSEK/VDX Fault-Tolerant Communication, Version 1.0*, July 2001. Available at http://www.osek-vdx.org/.
53. OSEK Consortium. *OSEK/VDX System Generation—OIL: OSEK Implementation Language, Version 2.5*, July 2004. Available at http://www.osek-vdx.org/.
54. International Standard Organization. *15765-2, Road Vehicles—Diagnostics on CAN—Part 2: Network Layer Services*. ISO, 1999.
55. OSEK Consortium. *OSEK/VDX Operating System, Version 2.2.2*, July 2004. Available at http://www.osek-vdx.org/.
56. P. Feiler. Real-time application development with OSEK—a review of OSEK standards, 2003. Technical Note CMU/SEI 2003-TN-004.
57. OSEK Consortium. *OSEKtime OS, Version 1.0*, July 2001. Available at http://www.osek-vdx.org/.
58. H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürstand, K.P. Schnelle, W. Grote, N. Maldenerand, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Voget, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and exploitation of the AUTOSAR development partnership. In *Convergence 2006*, Detroit, MI, October 2006.
59. S. Fürst. AUTOSAR for safety-related systems: Objectives, approach and status. In *Second IEE Conference on Automotive Electronics*, London, United Kingdom, March 2006.
60. P.H. Dezaux. Migration strategy of in-house automotive real-time applicative software in AUTOSAR standard. In *Proceedings of the 4th European Congress Embedded Real Time Software (ERTS 2008)*, Toulouse, France, 2008.
61. J. Sommer and R. Blind. Optimized resource dimensioning in an embedded CAN-CAN gateway. In *IEEE Second International Symposium on Industrial Embedded Systems (SIES'2007)*, Lisbon, Portugal, July 2007, pp. 55–62.
62. ASAM. *FIBEX—field bus exchange format, Version 3.0*. January 2008. Available at http://http://www.asam.net/.

63. G. Bauer and M. Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Nuremberg, Germany, 2000.
64. H. Pfeifer and F.W. von Henke. Formal analysis for dependability properties: the time-triggered architecture example. In *Proceedings of the 8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2001)*, Antibes, France, October 2001, pp. 343–352.
65. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis—the SymTA/S approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.
66. K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessors and Microprogramming*, 40:117–134, 1994.
67. N. Navet, Y. Song, and F. Simonot. Worst-case deadline failure probability in real-time applications distributed over CAN (controller area network). *Journal of Systems Architecture*, 46(7):607–617, 2000.
68. F. Simonot, F. Simonot-Lion, and Y.-Q. Song. Dependability evaluation of real-time applications distributed on TDMA-based networks. In *Proceedings of the 6th IFAC International Conference (FET'2005)*, Puebla, Mexico, November 2005.
69. X-by-Wire Project, Brite-EuRam 111 Program. X-By-Wire—Safety related fault tolerant systems in vehicles, final report, 1998.

5

FlexRay Protocol

Bernhard Schätz
Technical University of Munich

Christian Kühnel
Technical University of Munich

Michael Gonschorek
Elektrobit Corporation

5.1 Introduction

The introduction of electronic control units (ECUs) combined with embedded software in the automotive domain enables the implementation of complex functionalities like electronic motor management or electronic stability program, thus enhancing economics, security, and comfort of modern cars. To implement an advanced functionality like the stability program, it must interact with various other functionalities of the vehicle, like brake, engine, or gearbox control. Thus these functionalities require a suitable infrastructure for the distributed implementation of embedded control applications, using a network of communicating control units to exchange messages between them.

Due to their safety-critical character, the embedded automotive systems implementing these functionalities generally require guarantees about the properties of the communication infrastructure. Typical requirements are a guaranteed latency of transmission of (high-priority) messages and operational robustness of communication. Additionally, the economic constraints of the automotive domain require high efficiency, configuration flexibility, and low cost per communication node.

To combine operational and economic constraints, the ECUs are arranged in a network using a shared communication medium combined with a synchronization scheme to ensure reliable message exchange.

5.1.1 Event-Driven versus Time-Driven Communication

To provide a suitable infrastructure for ECUs, two different protocol paradigms have been provided:

- Event-driven communication, as, for example, implemented in the control area network protocol (CAN, see Chapters 4 and 6)
- Time-driven communication, as, for example, implemented in time-triggered protocol (TTP, see Chapters 4 and 15)

These paradigms differ fundamentally concerning the way in which access to the shared communication medium is coordinated between the communicating control units.

In the case of "event-driven communication," an ad hoc synchronization scheme is used. It is based on a *dynamic arbitration* policy avoiding access conflicts using a scheme based on message priority. This scheme does not require a priori coordination between the communication events of the processes; processes may send messages arbitrarily. Send access to the communication medium is granted on a per event basis: if the medium is not currently occupied, each process may try to access it to send a message. In case of a conflict between two or more senders, the sending of messages with lower priority is delayed in favor of the message with the highest priority. As a result, the protocol only deals with the treatment of communication events, requiring no further functionalities.

In contrast, in case of "time-driven communication," a predefined synchronization scheme is applied. It uses a *static arbitration* policy based on time slotting to avoid access conflicts. By a priori assigning a unique communication slot to each message, no ad hoc avoidance of conflicts is necessary. Access to the communication medium is granted based on a time-dependent strategy; a process may only send a message within the assigned time slot. However, conflict avoidance critically depends on the overall agreement of the start and duration of each time slot. Therefore, besides the actual communication, the protocol also has to include functionalities for the synchronization of the slots between all processes.

Due to the nature of their arbitration schemes, these two different paradigms offer rather complementary properties concerning the distributed development process and especially during the design and the integration phase. Event-driven communication uses a lightweight communication protocol. Thanks to its ad hoc conflict avoidance, no a priori reservation of time slots is necessary when designing the interface specification. Its synchronization strategy ideally allows optimal use of bandwidth even in case of sporadic messages. Depending on the nature of the communication load and the average—in practice lower than maximum—bandwidth, a maximum latency can be ensured for at least high-priority messages. Integration of components in the event-driven approach is achieved simply by composition; due to its synchronization strategy incompatibilities in the form of conflicts are resolved ad hoc. However, since latencies of messages depend on available bandwidth and communication load, latency guarantees established for a subset of components may not necessarily hold for the complete network of combined components.

In contrast, time-driven communication requires an additional clock-synchronization mechanism. Due to its arbitration strategy, time slots have to be reserved for all communication when designing the interface specification, possibly leading to unused slots and therefore loss of bandwidth especially in the case of rather sporadic messages. Using this a priori reservation scheme, time-driven communication ensures a maximum latency for all messages by construction through the explicit allocation of messages to time slots. Integration of components additionally requires ensuring the compatibility of the allocations to their time slots. However, latency guarantees established for components in isolation immediately hold for the composed system.

Thus, in total, the flexibility of event-driven communication concerning the integration and the accommodation of sporadic messages comes at the price of the composability of dependable systems. To amalgam both advantages, FlexRay combines both approaches.

5.1.2 Objectives of FlexRay

FlexRay was introduced to provide a communication system targeting specifically the needs of the high-speed control applications in the vehicle domain. Typical applications of this domain like advanced power train systems, chassis electronics, or by-wire functionality have rather different communication requirements, especially concerning the allowable latency of messages or their periodic/sporadic nature. Here, FlexRay is designed to support these different classes of applications by providing architectural flexibility through scalable functional communication alternatives.

To that end, FlexRay aims at providing a “best-of-both-worlds” approach by

- Integrating the event-driven and time-driven paradigm into a common protocol, to simultaneously offer both communication schemes
- Supporting the scalability of the ratio between the time-driven and event-driven parts of the communication

In the extreme case, FlexRay can be basically used as either a purely time-driven or a purely event-driven communication scheme; in practice, a mixture of both communication mechanisms is used. Additionally, FlexRay offers support to increase the reliability of communication by fail-safe mechanisms.

5.1.3 History of FlexRay

The FlexRay Protocol Specification V2.1 [FPS05] was defined in 2005 by the FlexRay consortium. The FlexRay consortium was formed out of a cooperation between BMW and DaimlerChrysler, looking for a successor to the current automotive standard

CAN and as an alternative to the communication standard TTP for future applications with high requirements concerning determinism, reliability, synchronization, and bandwidth, as needed for instance by X-by-wire functions.

The consortium was established in 2000 as an alliance of automotive, semiconductor, and electronic systems manufacturers. Its originating members in 2000 were BMW, DaimlerChrysler, Philips, and Motorola. Currently, the FlexRay consortium includes Bosch, General Motors, and Ford among others. While FlexRay kits became available in 2005, FlexRay was first introduced into regular production in 2006 by BMW in the suspension system of the X5 series.

5.2 FlexRay Communication

The exchange of information between nodes in a FlexRay cluster is based on a time-division multiple access (TDMA) scheme and organized in communication cycles, which are periodically executed from the startup of the network until its shutdown. One communication cycle is subdivided into time slots, in which the information transfer from one sender to one or more receivers takes place. The schedule of a FlexRay cluster determines in which time slots the FlexRay nodes are allowed to send their so-called frames. Sections 5.2.1 through 5.2.4 deal with the FlexRay frame format, the communication cycle-based media access strategy, as well as the static and the dynamic parts of a communication cycle.

5.2.1 Frame Format

As illustrated in Figure 5.1, a FlexRay frame consists of three segments: the header, the payload, and the trailer.

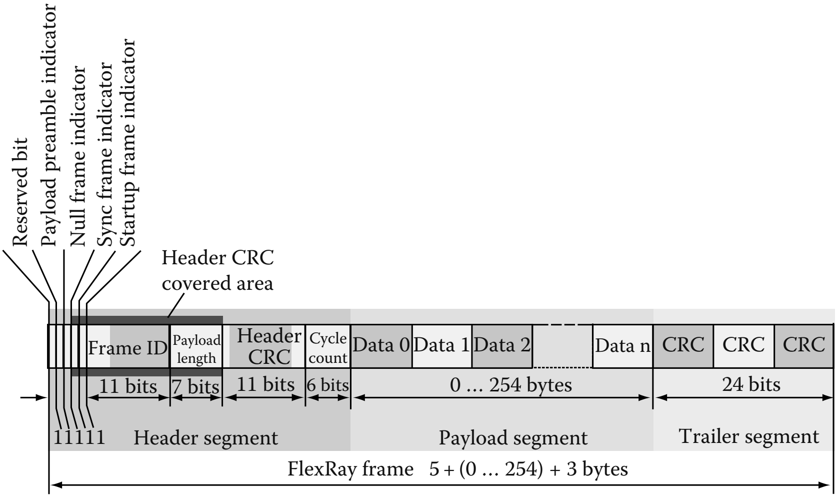


FIGURE 5.1 FlexRay frame format [FPS05]. (Copyright of Freescale Semiconductor, Inc. 2004, 2007. With permission.)

The header segment comprises the frame identifier, which the communication controllers use to detect errors, the payload length, the communication cycle counter, as well as the following protocol bits:

- The payload preamble indicator signals whether the optional vector for network management or extended message ID purposes lies in the first bytes of the payload segment or not.
- The null frame indicator determines if the frame contains usable data.
- The sync frame indicator marks whether the frame is used by the communication controllers for the global clock synchronization.
- The startup frame indicator signals if the frame is involved in the FlexRay startup procedure.

The payload segment contains the effective information that is to be exchanged. The amount of payload data contained in a single FlexRay frame ranges from 0 to 254 bytes.

The complete frame is protected by the trailer segment, which consists of a 24 bit frame cyclic redundancy check (CRC). Parts of the header segment are additionally covered by an 11 bit header CRC with a Hamming distance of 6.

5.2.2 Communication Cycle

The FlexRay communication takes place in communication cycles that have a predefined length. As can be seen in Figure 5.2, a FlexRay communication cycle is comprised of a compulsory static and an optional dynamic segment as well as one or two protocol segments, namely the mandatory network idle time (NIT) and the optional symbol window. The static segment consists of a certain number of static slots with the same fixed duration. The dynamic segment consists of so-called minislots, which can be used by the host for transmitting frames with a variable payload length, sporadic frames, or frames with a period higher than the communication cycle length. The slots of the two segments are consecutively numbered, starting at one with the first static slot.

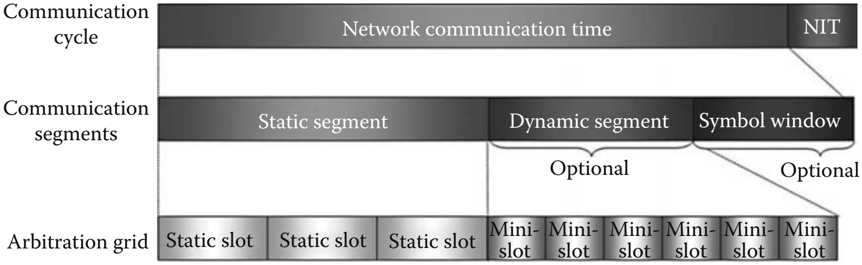


FIGURE 5.2 FlexRay communication cycle. (Copyright by DECOMSYS, Member of Elektrobit. With permission.)

The symbol window is used for network management purposes only. It is a time slot in which the media access test symbol (MTS) can be transmitted over the network.

In the NIT the bus is free of communication; the NIT is used by the communication controllers to perform the clock-synchronization algorithm. The length of the NIT has to be configured at system design time. It is crucial that the NIT be long enough for all involved communication controllers to be able to handle their calculations.

A minimal configuration of a FlexRay communication cycle must contain a static segment and the NIT. A FlexRay cluster has to consist of at least two nodes and consequently a minimum of two static slots are required for synchronization.

The FlexRay protocol supports a differentiation of 64 communication cycles by using a global 6 bit cycle counter. The cycle counter value is transmitted in each frame header. This differentiation allows a node of a cluster to transmit different frames in the same slot in different communication cycles (cycle filtering). Besides, the so-called slot or cycle multiplexing is possible in the dynamic segment, that is, different nodes can send frames in the same slot but in different cycles. In addition to the global FlexRay time base, the host software can use the cycle counter for synchronizing software routines with specific time slots in order to minimize signal latencies.

5.2.3 Static Segment

In the static segment of a communication cycle, all slots have the same fixed duration. FlexRay supports up to two FlexRay channels, and each slot is exclusively owned by one communication controller per channel for transmission of a frame. If two channels are provided in a cluster topology, a controller can use a slot either for a redundant transmission of one frame or for a transmission of two different frames. A third possibility is that two different communication controllers use the slot for frame transmission, one on each channel. Figure 5.3a illustrates a FlexRay cluster topology and Figure 5.3b gives an example of communication in the static segment during one communication cycle. Nodes A and C use slots 1 and 3 on both channels for a redundant transmission of a frame whereas node B is only connected to channel b and transmits its frames in slots 2 and 4. Node D also uses slot 4 for the transmission of a frame, but on channel a. Nodes C and E share slot 7 for the transmission of their frames. In slot 5, node A transmits two different frames; slots 6, 8, 9, and 10 are not used.

Not every time slot has to be allocated to a frame; under certain conditions, it makes sense to reserve a certain bandwidth for nodes that will be integrated into the cluster at a later point in time. If a static slot is not allocated to a frame in a communication cycle, it remains empty and its bandwidth is wasted. In contrast, a frame that is assigned to a time slot will always be sent by the corresponding communication controller.

This fixed reservation of time slots that are dedicated to communication controllers brings the advantage of strong guarantee of the message latency. It is exactly known when a specific frame will be transmitted on a channel, and because the communication is collision-free, the worst-case transmission time can be calculated.

FlexRay requires a specification of the communication schedule at the design stage. The schedule holds information that is essential for the communication controllers, for example, in which slots they will send or receive their frames. With respect to frame transmission in the static segment, important parameters for the configuration

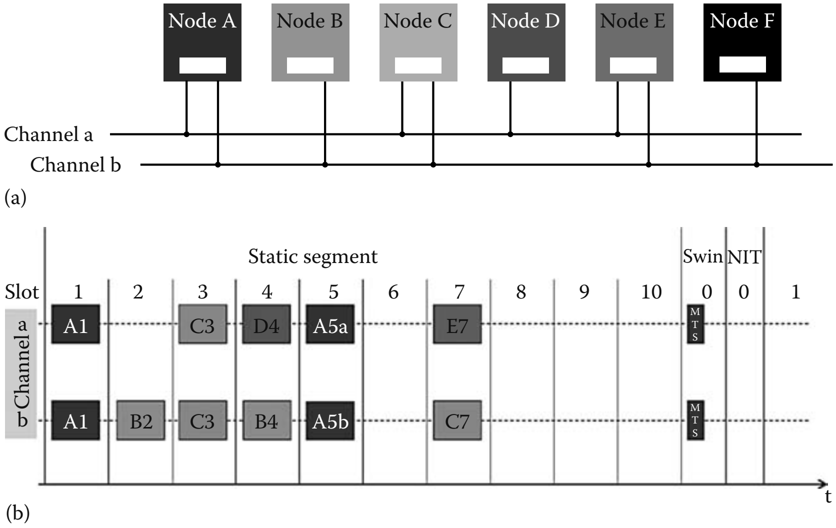


FIGURE 5.3 Example of communication in the static segment. (a) Sample cluster. (b) Frame transmissions in several static slots. (Copyright by DECOMSYS, Member of Elektrobit. With permission.)

of the communication controller for frame transmission in the static segment include the identifier of the time slot (Slot ID), the channel, and the cycle filter information.

The FlexRay low-level parameters, which describe the significant FlexRay protocol properties, have to be specified at system design time as well. Examples of these parameters are the quantity and the duration of the static slots, the length of the communication cycle, and the action point offset, that is, the delay of a frame transmission relating to the start of the time slot. The parameters are part of the network configuration and have to be identical in all communication controllers that participate in the communication of a particular FlexRay cluster. For each communication controller more than 50 parameters have to be configured by the user at system design time. To simplify this complex task, tool support is highly recommended.

Besides the full deterministic communication timing, an important feature of the static segment is its composability. This property plays an important role in the series productions of car manufacturers (original equipment manufacturers [OEMs]). When the subsystems of OEMs and their suppliers are integrated into the complete system, the behavior of the subsystems is not influenced because the fixed time-dependent behavior in the static segment prevents the communication properties of the subsystems from changing. Figure 5.4 exemplifies FlexRay composability. The integration of the subsystems of supplier 1, supplier 2, and the OEM does not change the communication timing, that is, the position of the static time slots.

5.2.4 Dynamic Segment

In the dynamic segment of a communication cycle, a more flexible media access control method is used: the so-called flexible TDMA (FTDMA) scheme. This scheme,

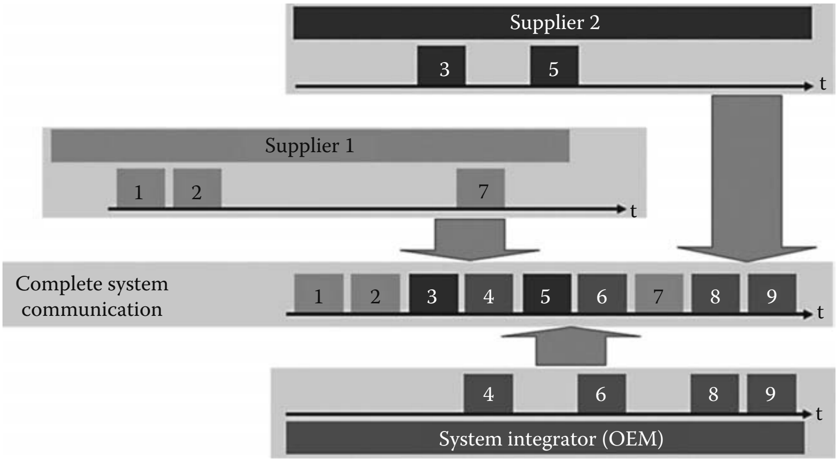


FIGURE 5.4 Example of FlexRay composability. (Copyright by DECOMSYS, Member of Elektrobit. With permission.)

which is based on the byteflight protocol [BFL] developed by BMW, is priority and demand driven. The dynamic part is subdivided into minislots, which have only a short duration. Similar to the static slots, these minislots can be assigned to frames, but the transmission of a frame will only be started if the controller has data to send. Hence, the decision whether a frame is transmitted in the dynamic segment is made by the host software during runtime. If not a single dynamic frame is transmitted, the dynamic part of the communication cycle is fully unused (Figure 5.5a).

The duration of the minislots is not long enough to accommodate a total frame transmission. In case a minislot is not used for communication, only a small amount of bandwidth will be wasted. If a communication controller decides to transmit a frame in a minislot, however, the minislot is expanded to the size of an adequate time slot (e.g., slot 4 on channel a in Figure 5.5b). The payload length of dynamic frames is not predetermined and can be changed by the host software during runtime. It is only limited by the buffer size of the communication controllers. With the expansion of a minislot, the number of the available minislots in the dynamic segment is reduced. The more frames are sent in the early stages of the dynamic part, the smaller is the chance of a frame transmission at a late point in the dynamic segment. Thus, the transmission of frames in the dynamic segment is priority driven; a frame in a minislot with a lower identifier has a higher priority. A frame that was requested for transmission but could not be sent in a communication cycle will be sent by the controller in the same slot at the next opportunity.

The deferral of the minislots in the cycle time, which depends on the number and the length of transmissions in previous dynamic time slots, leads to a logical subdivision of the dynamic segment into two parts. In the guaranteed dynamic segment, all scheduled frames will be transmitted independently of the bus load. In the rest of the dynamic segment, a transmission of a frame in the actual communication cycle is not assured. Here the communication controller will only start the transmission if

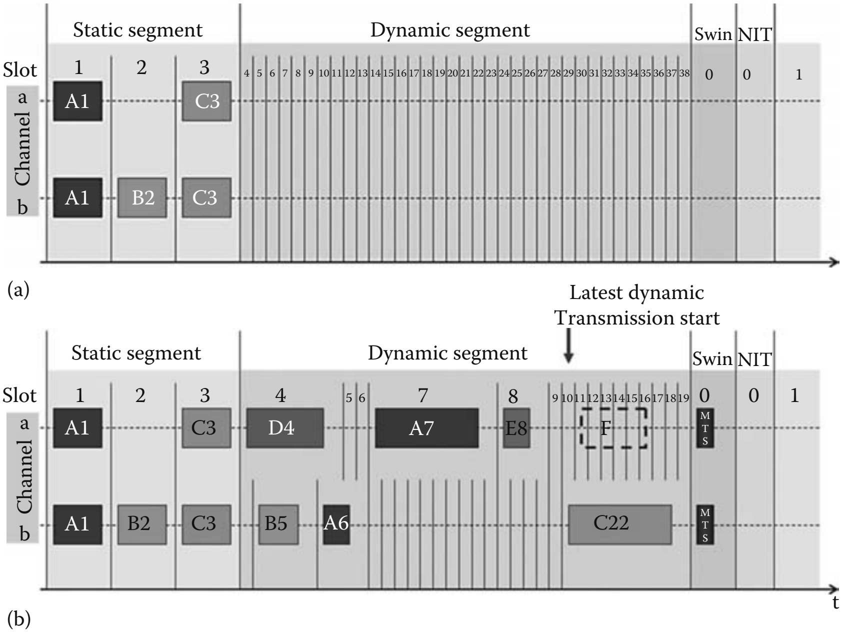


FIGURE 5.5 Example of communication in the dynamic segment. (a) No frame transmissions in the dynamic slots. (b) Several frame transmissions in the dynamic slots. (Copyright by DECOMSYS, Member of Elektrobit. With permission.)

the latest dynamic transmission start point, a protocol parameter that has to be predefined by the user, has not been passed. In the example shown in Figure 5.5b, the frame F, which should be transmitted on channel a in slot 11, will not be transmitted in the current cycle because of too much bus load in previous time slots of the current dynamic segment.

5.3 FlexRay Protocol

5.3.1 Protocol Architecture

In FlexRay a “cluster” is a set of “nodes” connected by one or two “channels.” All nodes of a cluster have a common global clock. A node consists of a host and a FlexRay controller. The controller is responsible for the communication between the nodes, whereas the host runs the operating system and application software. The physical interconnection of a cluster may be a bus or star topology or even a mixture thereof. The architecture of a FlexRay node is illustrated in Figure 5.6. In the context of FlexRay, the applications, OS/middleware, and the microcontroller (indicated in gray) are called “host”, the FlexRay controller (indicated in black) is simply called “controller” and the combination of host, controller, and the optional bus guardians (labeled BG, see Section 5.3.4) is called “node.”

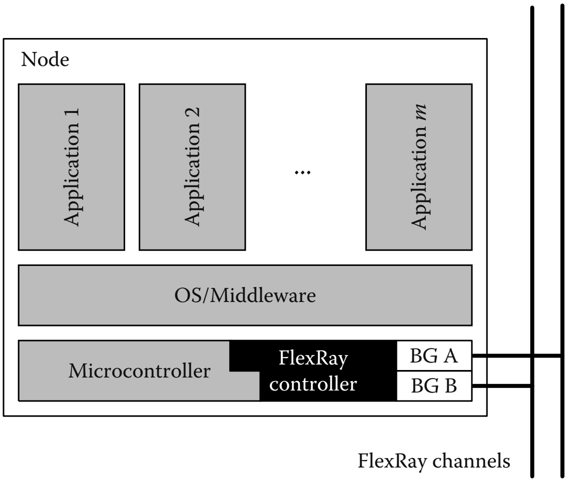


FIGURE 5.6 Architecture of a FlexRay node.

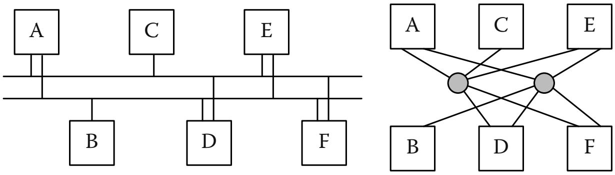


FIGURE 5.7 Bus topology (left) and active star topology (right).

There are two basic topologies available for FlexRay as illustrated in Figure 5.7: bus and active star. In the "bus topology" all nodes are connected to the same physical cable(s), whereas in the "active star topology," every node is connected directly to the star couplers. In both cases a node may be connected to either one of the channels or both. A cluster may also employ a mixture of bus and star topologies. In both topologies, the ends of the cable have to be terminated by a resistor.

5.3.2 Protocol Wakeup and Startup

Before a cluster can commence with its communication as described in Section 5.2, it has to be initialized. In FlexRay, the initialization of a cluster consists of two phases: "wakeup" and "startup." In the wakeup phase the nodes of the cluster are powered up and the hosts boot their operating system. After synchronizing their clocks in the startup phase the nodes enter normal operation. The startup and wakeup phases will be described in more detail in the following.

During wakeup and startup some nodes take a special role: wakeup nodes and coldstart nodes. A "wakeup node" is a node that may wake a sleeping cluster by sending a wakeup pattern. A "coldstart node" is a node sending startup frames during the startup phase. A wakeup node is not required to be a coldstart node or vice versa.

In the following only the nonerror case is discussed, for information about the behavior in presence of error consult the specification [FPS05].

5.3.3 Wakeup

The wakeup of a cluster is depicted in Figure 5.8. When the cluster is asleep and the first node of the cluster is woken by an external event, it boots its OS and configures the FlexRay controller. Only one channel is woken at a time to prevent a faulty host from disturbing the communication of an already running cluster on both channels. So the host has to select the channel to be woken. The host notifies the controller to send the "wakeup pattern" on this channel. Once the other controllers have received this wakeup pattern they wake their hosts. One of this newly waken nodes may then wake the second channel, which works just the same way as with the first channel. As at least two nodes are necessary to start up the cluster, the first node waits a predefined amount of time, in which at least one other node is assumed to be ready for startup. Then the waken nodes enter the startup phase.

5.3.3.1 Startup

In the startup phase, as illustrated in Figure 5.9, there are three different types of nodes: exactly one leading coldstart node, at least one following coldstart node, and any number of non-coldstart nodes. A coldstart node becomes the "leading coldstart node," if it is in startup mode and does not receive any communication. If it does receive communication it becomes a "following coldstart node" as it assumes that there already is a leading coldstart node.

The leading coldstart node is the first to send a collision avoidance symbol (CAS) and after that starts with the first communication cycle by transmitting its "startup frame." The CAS is used to detect if more than one coldstart node are attempting a startup. Each coldstart node has exactly one startup frame, a frame with the startup and sync bits set in the header segment (Section 5.2.1). After four cycles the leading coldstart node receives the startup frames of the following coldstart nodes. If the clock synchronization with the other coldstart nodes is successful it enters normal operation.

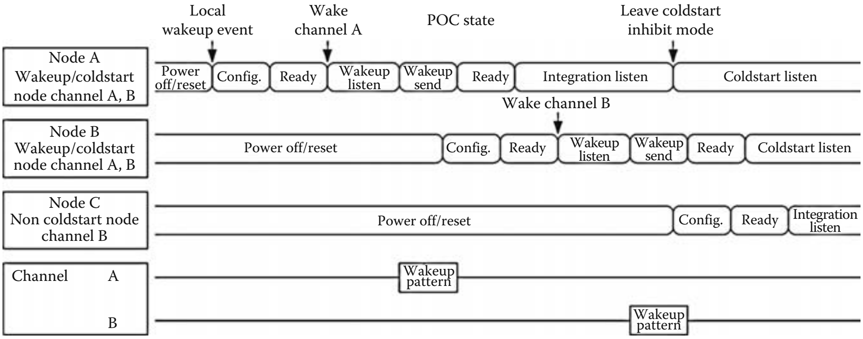


FIGURE 5.8 Cluster wakeup [FPS05]. (Copyright by NXP. With permission.)

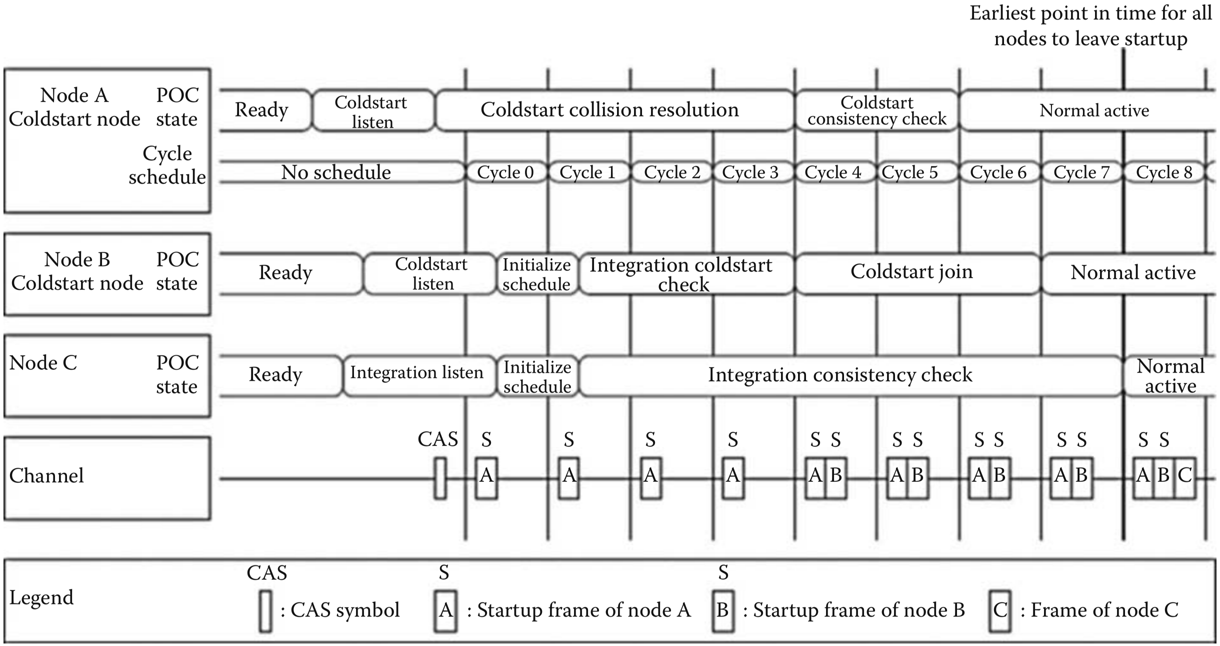


FIGURE 5.9 Cluster startup [FPS05]. (Copyright by NXP. With permission.)

A following coldstart node waits until it receives two consecutive frames from the leading coldstart node. These two frames are required to construct an initial schedule. If successful, it collects all sync frames from the next two cycles and performs the clock synchronization. After a successful initialization of the clock synchronization it starts to send its startup frame. If there are no errors for three more cycles, the following coldstart node enters normal operation.

A "non-coldstart node" waits until it receives two consecutive frames from the leading coldstart node. It performs the clock synchronization on those two frames for an initial schedule. For the following four cycles it collects all sync frame and performs the clock synchronization. If successful, it enters normal operation. The main difference between non-coldstart nodes and following coldstart nodes is that the non-coldstart node does not send any frames during startup.

Following this schema, all nodes leave startup at the end of cycle 7 if no error occurs. The startup of a node is identical to the "reintegration" of a node into a running cluster.

5.3.4 Clock Synchronization

In a TDMA network such as FlexRay, all nodes of a cluster need a common global clock so that every node will only send in its time slot(s). At the startup of the system all local clocks of the nodes have to be synchronized to provide this global time. Each node has a local clock generator, usually based on the resonance frequency of a quartz crystal. As two crystals rarely have exactly the same resonance frequency, the local clock start drifting apart during operation. A typical quartz crystal has a precision in the magnitude of 50 ppm. The frequency of a crystal cannot be determined statically, since it is also influenced by external factors such as temperature and vibration [WG92]. So the local clocks have to be synchronized regularly during operation of the system.

As perfect synchrony is impossible to achieve in a distributed system, the global clock is not the same on all nodes, but the difference between the local clocks has an upper bound. This bound is assumed to be low enough for the application domain, so a common global time can be assumed on some level of abstraction. In the following, the timing mechanisms of FlexRay are introduced, followed by the clock-synchronization algorithm.

5.3.4.1 Timing in FlexRay

Time in FlexRay is defined by a triple consisting of cycle, macrotick, and microtick counters, where a cycle consists of several macroticks and a macrotick of several microticks as depicted in Figure 5.10. The number of macroticks per cycle is constant for the entire cluster, as the macroticks are synchronous on all nodes. The microticks are generated by the local clock generator, thus the current value of microticks is only valid within one node. The number of microticks per macrotick depends on the frequency of the local clock and is also individual for each node. This number is adjusted during runtime to synchronize the clocks of the different nodes.

The clock-synchronization algorithm of FlexRay is an extension of the Welch–Lynch algorithm [WL88]. The local clock of the host may be synchronized with

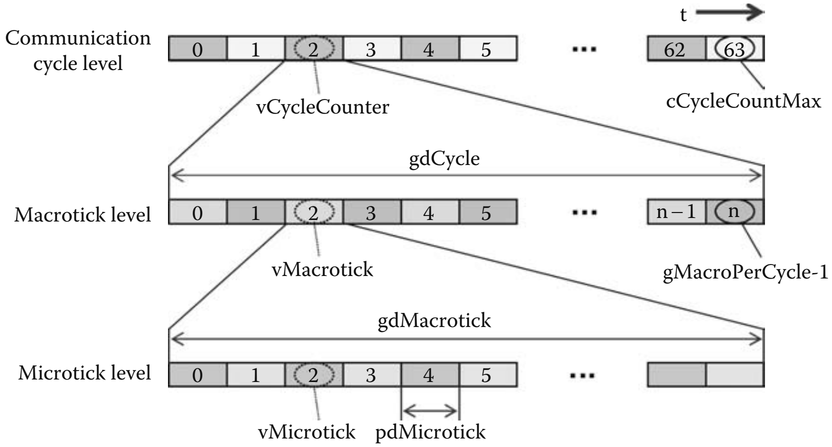


FIGURE 5.10 Timing hierarchy [FPS05]. (Copyright of Freescale Semiconductor, Inc. 2004, 2007. With permission.)

the global time of FlexRay so that applications running on the host also have a cluster-wide synchronous notion of time (Section 5.4.1). The global clock may also be synchronized between two clusters or based on an external time source.

The measurement of the timing is performed only during the static segment and the computation of the correction values during the dynamic segment and/or NIT (Section 5.2.2).

5.3.4.2 Measurement of Clock Drift

A node can compute the timing error between its local clock and the individual clocks of the other nodes by observing the arrival times of the frames of the other nodes. This way no additional communication is needed.

Whenever a node receives a sync frame from another node, a timestamp is stored along with the frame. A sync frame is a frame with a set sync bit (Section 5.2.1); nonsync frames are not considered for the clock synchronization. The deviations are stored for all sync frames separately of even and odd cycle numbers. The clock correction values are computed based on these measurements. An example of the results of the measurement is presented in Table 5.1.

TABLE 5.1 Example of Measurements with Difference

Frame	Sync Bit	Expected	Even Cycle Measurement	Odd Cycle Measurement	Offset Difference	Rate Difference
1	+	10	11	12	2	1
2	+	20	19	18	−2	−1
3	−	30	33	33		
4	−	40	41	39		
5	+	50	52	51	1	−1

5.3.4.3 Calculation of Correction Values

FlexRay uses a combination of two different correction values: offset correction and rate correction. Offset correction is used to reduce the current deviations of the clocks, while rate correction is used to anticipate the drift rate of the clocks, as depicted in an example in Figure 5.11 with four local clocks. The gray vertical lines indicate the application of the clock correction.

To compute the correction values the "fault-tolerant midpoint algorithm" [FPS05] is used. The algorithm takes a list of numbers as input and delivers the midpoint as output. An example of this algorithm is depicted in Table 5.2. From the initial sorted list (1), the k largest and k smallest numbers are removed (2) thereby excluding k possibly faulty nodes from influencing the clock synchronization. The parameter k is chosen, depending on the number of input values: for 1 or 2 input values $k = 0$, for 3 to 7 input values $k = 1$, and for more than 7 input values $k = 2$. After the elimination, the algorithm takes the largest and smallest elements of the remaining list (3) and returns the average of those two (4).

In every odd communication cycle the "offset correction" value is calculated. For each sync frame received during this cycle, the difference between expected arrival time and actual arrival time is computed. The fault-tolerant midpoint algorithm is applied to this list and delivers the offset correction value. If the correction is within certain bounds it will be applied at the end of the cycle. In the example in Table 5.1 the

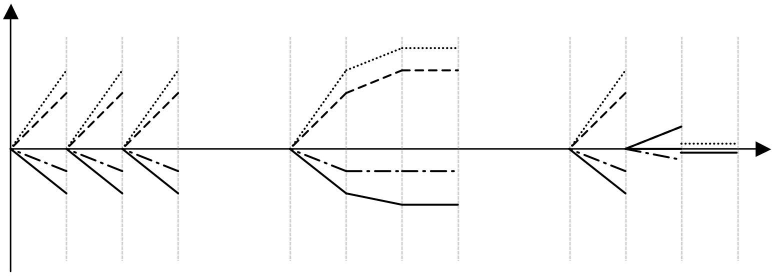


FIGURE 5.11 Illustration of the clock-synchronization algorithm offset correction (left), rate correction (center), combined rate and offset correction (right).

TABLE 5.2 Example of Fault-Tolerant Midpoint Algorithm

1. Input Values	2. Remove k=2 Smallest and Largest	3. Take Smallest and Largest	4. Build Average
−25			
−12			
−7	−7	−7	
−3	−3		
0	0		(−7 + 5) : 2 = −1
4	4		
5	6	5	
8			
19			

column "offset difference" is the difference between the columns "expected" and "odd cycle measurement." The resulting offset correction value in this example would be 1.

The "rate correction" value is computed in every odd communication cycle based on the measurements of the current and the previous even cycle. For each measured sync frame the difference between the deviation values of this cycle and the last cycle is computed. This delivers a drift rate for each of the other nodes. The fault-tolerant midpoint algorithm is applied to these drift rates and delivers the rate correction value for this node. Again if the correction value is within certain bounds, it will be applied at the end of this cycle. In the example in Table 5.1 the column offset difference is computed as the difference between the columns "even cycle measurement" and "odd cycle measurement." The rate correction value for that example is −1.

5.3.4.4 Application of Correction Values

The current correction values are applied during the NIT of the current cycle. The offset correction is performed every odd cycle, the rate correction during the following even cycle.

The offset correction value is simply added to the NIT and thereby shortens or extends the overall length the communication cycle once as depicted in Figure 5.12.

The rate correction value is added to the number of microticks per cycle. As the number of macroticks per cycle is static, the number of microticks per macrotick is computed based on the new number of microticks per cycle. The rate correction value thereby influences the length of the next two cycles, where it is newly evaluated.

5.3.5 Fault-Tolerance Mechanisms

FlexRay offers several other mechanisms for fault-tolerance, apart from the one implemented in the clock-synchronization algorithm, to compensate for other communication faults.

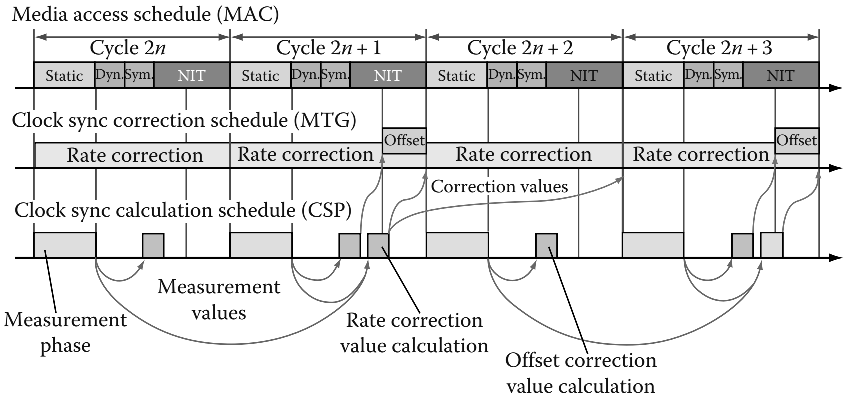

Media access schedule (MAC)
Cycle 2n
Cycle 2n + 1
Cycle 2n + 2
Cycle 2n + 3
Static
Dyn.
Sym.
NIT
Clock sync correction schedule (MTG)
Rate correction
Offset
Correction values
Clock sync calculation schedule (CSP)
Measurement values
Measurement phase
Rate correction value calculation
Offset correction value calculation

FIGURE 5.12 Application of correction values [FPS05]. (Copyright of Freescale Semiconductor, Inc. 2004, 2007. With permission.)

One strategy is redundancy. A cluster may use two separate communication channels for communication. By using a different physical topology for both channels a physical fault is less likely to disturb the whole cluster. When using two channels, a message may be sent separately on both channels in different slots. So a transient error will only lead to a belated rather than a lost message.

As most communication protocols, FlexRay uses a CRC to cope with bit errors on the bus, in this case a 24 bit CRC is used. The probability of undetected network errors in this case is less than 6×10^{-8}. At 10,000 messages per second and an estimated rate of 10^{-6} for uncorrelated bit errors, this means approximately 2×10^{-6} undetected erroneous frames per hour [PMH05]. This in turn means that only 1% of all vehicles will ever experience an undetected erroneous frame during an average life span of 6000 h of operation [KS06].

FlexRay uses fault containment in several areas to ensure that a local fault of a node does not disrupt the communication of the remaining cluster. One prominent fault containment method is the bus guardian. It observes one or more controllers, depending on topology, and only allows a controller to transmit on the bus during one of its assigned time slots. Outside of its slots, the bus guardian will filter all transmissions of the controller. This ensures that a "babbling idiot" will not disrupt the communication of the intact nodes. In FlexRay, there are two possibilities of placing the bus guardians: "node local" bus guardians are located on each node, as depicted in Figure 5.6 whereas "central" bus guardians are integrated into the star couplers as shown in Figure 5.7.

For safety-critical application further mechanisms might be required, such as message acknowledgment and a membership vector. As these are not part of FlexRay they have to be implemented in a higher protocol layer.

5.4 FlexRay Application

This section focuses on software components and implementation strategies that are necessary and useful for running a FlexRay application. In addition, it provides a short overview of possible supporting tools for software developers and test engineers.

5.4.1 FlexRay Implementation

FlexRay drivers and related communication layers offer the application developer an easy-to-use frame or signal-based interface to the FlexRay communication system. These software modules are typically configured by network design tools. Currently there are several FlexRay drivers and configuration tool solutions on the market, differing in performance, usability, functional range, and price. In the future, however, the AUTomotive Open System Architecture (AUTOSAR; see Chapter 2) [AUT] standard will become increasingly relevant in projects of car manufacturers. Among others, a FlexRay driver and a FlexRay interface module are specified in AUTOSAR and will become available for a growing number of hardware platforms.

The development of a FlexRay application requires several elementary decisions concerning the system architecture. Three scenarios regarding the synchronization between the FlexRay global time and the application are possible:

- The application and the FlexRay communication are executed asynchronously
- The application is synchronized to the FlexRay global time
- Parts of the application are running synchronously to the FlexRay global time

The best results with respect to signal latencies can be achieved if the application runs synchronized to the FlexRay global time. In this case, the application of a transmitting node can send up-to-date data to the communication controller shortly before the time slot in which the accordant frame transmission starts. The application of a receiving node can read the data from its communication controller immediately after the frame has been received, provided that it is also synchronized to the FlexRay global time. An example of this can be seen in Figure 5.13. This short, fixed, and guaranteed latency of the data from the sending task in one ECU to the receiving task in another ECU allows the creation of high-level distributed control systems. Because of this determinism, FlexRay is particularly convenient for reliable safety subsystems or driving dynamics systems. Together with the option of communication redundancy on two channels, FlexRay is furthermore appropriate to the creation of fault-tolerant X-by-wire systems.

The synchronization between the application and the FlexRay communication can be achieved either by using interrupts from the communication controller or by means of a real-time operating system that supports external time synchronization mechanisms (e.g., OSEKtime).

For various reasons, there are cases in which synchronization between the application and the FlexRay communication is not possible or not required. In a motor control unit, for example, the software has to run synchronously to the motor rotation speed and cannot be adapted to another time base. In cases like this, only communication controllers that support this asynchronous operation should be used. But even then it is not possible to exercise control on the behavior of the system regarding exact signal latencies. If, for instance, an application task writes data to a communication controller shortly after the accordant time slot has passed, the communication

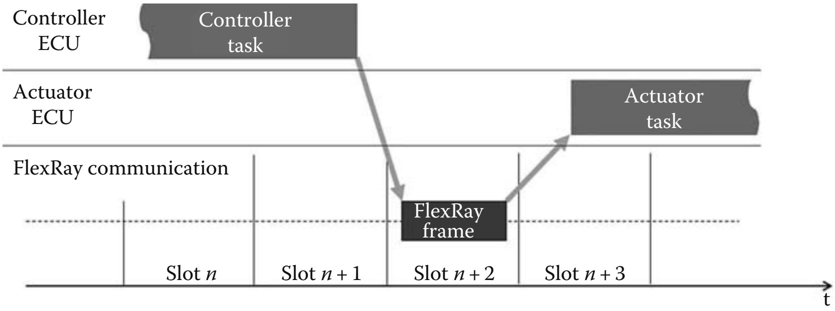


FIGURE 5.13 Exemplary control system with FlexRay. (Copyright by DECOMSYS, Member of Elektrobit. With permission.)

controller has to "wait" for the next time slot that is assigned to the frame. In which communication cycle the data will finally be sent cannot be predicted; "only a value for the worst-case latency can be determined." If a signal group is spread over more than one frame, it is not possible to guarantee consistency among the signals. Hence, an application that is completely running asynchronously is not recommendable.

If the application has to run asynchronously to the FlexRay communication, another option is to synchronize only FlexRay-related communication layers to the FlexRay global time. Particularly the FlexRay communication tasks, which are responsible for the data transfer between the application and the communication controller, will be synchronized in such cases and will thereafter perform the temporal decoupling.

5.4.2 FlexRay Tool Support

The manual configuration of the nodes in a FlexRay cluster without tool support is time consuming and error prone. A faulty schedule or node configuration in a FlexRay cluster generally necessitates a complex and expensive error analysis. For this reason, tool support is indispensable in the majority of cases.

In general, FlexRay network design tools allow the specification of the hardware architecture of the nodes, the planning of the communication schedule, and the software task schedule as well as the generation of the node configuration. In addition to this, some of the tools support the OEM/supplier development process. The OEMs develop distributed applications and plan a network of ECUs, functions, a communication schedule, and the global FlexRay parameters. Afterward they provide the ECU suppliers with all the information that is relevant to their particular ECU. The ECU suppliers import this information into a configuration tool, extend this configuration by ECU specific information, and generate the source code configuration files for the ECU, all with very little effort. For data exchange, extensible markup language (XML) files are typically used.

FlexRay monitoring and analysis tools are also very important for distributed application development. These tools are typically used for monitoring and logging the network startup and ongoing communication. In the case of a fault, the analysis tool provides detailed status and communication information to the FlexRay user and possibly even a trigger output for further measurement devices (e.g., an oscilloscope, which is used to capture a destroyed FlexRay frame for an advanced analysis). With the objective of finding logical faults in distributed systems, the tools are used to simultaneously monitor different communication systems and hardware interfaces, such as CAN, FlexRay, digital, and analog signals.

For the development and the testing of an ECU, it is not always possible to provide all other ECUs of the cluster as communication partners. Even for the FlexRay network startup, the existence of at least one startup partner is required. Besides, it is often necessary to stimulate the ECU with applicable data to prevent it from entering an error state. For this purpose, various cluster simulation solutions are already available, ranging from a simple "startup buddy" to hardware-in-the-loop test benches for FlexRay.

5.5 Conclusion

FlexRay is intended to provide an advanced communication technology for automotive high-speed control applications in power train, chassis, and body control, with availability, reliability, and bandwidth as key features of the system, to increase safety, reliability, and comfort of vehicles. This requires the "provision" and "application" of the guaranteed availability, reliability, and bandwidth. Thus, the introduction of FlexRay as communication technology also raises questions concerning its impact on the development process as well as the verification of the intended properties, besides the technical features of the protocol discussed in the previous sections.

5.5.1 Impact on Development

FlexRay—due to its hybrid character—combines aspects from the time-driven and the event-driven interaction paradigm, the former adopting a viewpoint of (predefined) periodic interaction, the latter a (undefined) sporadic one. Like in other domains of embedded systems, both the event-driven and the time-driven points of view have their points in the construction of automotive systems, due to the conflicting goals of efficiency of resources usage (e.g., unused statically reserved processing/communication slots) and reliability of functionality (undefined latency of process execution/message transmission).

When describing the functionality of the control processes, there are tasks that are naturally described as either time- or event-driven. For instance, ignition control in motor management has subtasks of each domain. Controlling the timing of the ignition depends on the position of the engine, defined by an (sporadic) event characterizing the zero-position of the flywheel; computing the amount of fuel injected depends on the amount currently requested by the position of the accelerator pedal, defined by a maximum (periodic) time characterizing the validity of the request.

Similarly, when considering communication, messages addressing the configuration, maintenance, and diagnosis of a system are of a sporadic, event-driven nature, while messages addressing the control of continuous physical processes are of a periodic, time-driven nature.

In contrast to the scheduling of processing tasks, due to the available bandwidth of communication networks including different modes of operations and the allowable latency and period of globally distributed signals, the efficiency of bus usage in many cases becomes less important than the reliability of transmission when scheduling communication tasks.

Two other conflicting development goals affected by the choice between a time-driven and an event-driven approach to system development are "flexibility" and "safety" of system design. To ensure safety aspects of a system, guarantees about properties of the system have to be established (e.g., maximum latency between the activation of a crash sensor and the firing of an air bag). The predictability of the overall system is greatly facilitated by strong (i.e., deterministic and static) guarantees provided by its components (e.g., deterministic and static processing and communication latencies). Domains like avionics favoring safety over flexibility explicitly recommend static deterministic scheduling of processes and interactions. On the other hand, to enable flexibility of a system architecture, local changes in a system must be possible (e.g., using variants of a component with different interaction patterns). The changeability of the overall system is greatly facilitated by liberal (i.e., nondeterministic and flexible) assumptions made by its components (e.g., timing of message exchange).

However, experiences show that safety properties not ensured by construction through a restrictive design often cannot be established by verification, especially when considering the typically large of variants of control units in the automotive domain. To support a "modular" development process of "reliable" systems, a precise description of the component interfaces is indispensable. Architecture-oriented approaches (e.g., AUTOSAR) therefore explicitly address the importance of interface descriptions capturing the (temporal) characteristics of the exchanged signals.

The assignment of signals to segments (i.e., static/dynamic) depends on the functional character of the signals. Signals related to controlled process generally have a maximum validity, suggesting a static schedule: the frequency of the signal depends on the characteristics of its change (e.g., the engine speed needs a higher frequency of update than the status of door lock). Signals not related to controlled processes (e.g., read-out of diagnostic information, change of configuration parameters) are typical candidates for the dynamic segment.

Due to its hybrid character, FlexRay offers a range of different configuration options, with a CAN-like configuration using only the dynamic part, as well as a TTP-like configuration using only the static part, as well as all variations in between. Thus naturally, it can be used, for example, as a mere substitute for a CAN-based communication. Most importantly, due to its flexibility, it can also be used for a smooth migration process from an event-driven to a time-driven system architecture.

Besides FlexRay, there also exist other combinations of event-driven and time-driven communication paradigms (e.g., TTP with CAN emulation, or TTCAN), providing similar characteristics but differing concerning the emphasis on the time- or event-triggered paradigm.

5.5.2 Verification of FlexRay

As FlexRay is a relatively new technology, developed by an industrial consortium, the amount of published verification efforts is very limited. The interaction between a FlexRay controller and an OSEK FT-COM communication layer in analyzed in Ref. [KS06]. A verification of a reduced version of FlexRay is planned in the Verisoft project [VER]. A schedulability analysis for the static and dynamic segments is presented in Ref. [TPP06]. The startup behavior was model checked in Ref. [SK06] and some issues were found there. A plan to verify the clock-synchronization protocol was announced in Ref. [BPT05], but results have not been published yet. Rushby [RUS02] presents an overview of the verification of the time-triggered architecture (TTA). Although the TTA is a similar approach (Chapter 15), designed against a similar set of requirements, the differences in the realization are quite fundamental so the results cannot be transferred to FlexRay directly, but the verification approaches of TTA might be reused for FlexRay.

References

[AUT] AUTOSAR Development Partnership, www.autosar.org

[BFL] Byteflight protocol, www.byteflight.com.

[BPT05] D. Barsotti, L. Prensa Nieto, and A. Tiu. Verification of clock synchronization algorithms: experiments on a combination of deductive tools. In: *Proceedings of the Fifth International Workshop on Automated Verification of Critical Systems*, Coventry, UK, 2005.

[FPS05] FlexRay Consortium. FlexRay Communications System Protocol Specification Version 2.1, 2005.

[KS06] C. Kühnel and M. Spichkova. Upcoming automotive standards for fault-tolerant communication: FlexRay and OSEKtime FTCOM. In: *Proceedings of International Workshop on Engineering of Fault Tolerant Systems (EFTS 2006)*, Luxembourg June 12–14, 2006.

[PMH05] M. Paulitsch, J. Morris, B. Hall, K. Driscoll, E. Latronico, and P. Koopman. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In: *Proceedings of International Conference on Dependable Systems and Networks*, Yokohama, Japan, 2005.

[RUS02] J. Rushby. An overview of formal verification for the time-triggered architecture. In: *Proceedings of Seventh International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, Oldenburg, Germany, 2002.

[SK06] W. Steiner and H. Kopetz. The startup problem in fault-tolerant time-triggered communication. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN06)*, Philadelphia, PA, 2006. http://www.dsn2006.org.

[TPP06] T. Popp, P. Popp, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS 06)*, Dresden, Germany, 2006.

[VER] Verisoft Project. www.verisoft.de.

[WG92] F.L. Walls and J.-J. Gagnepain. Environmental sensitivities of quartz oscillators. *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 39 (2), 241–249, 1992.

[WL88] J. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77 (1), 1–36, 1988.

6

Dependable Automotive CAN Networks

Juan Pimentel
Kettering University

Julian Proenza
University of the Balearic Islands

Luis Almeida
University of Aveiro

Guillermo Rodriguez-Navas
University of the Balearic Islands

Manuel Barranco
University of the Balearic Islands

Joaquim Ferreira
Polytechnic Institute of Castelo Branco

6.1 Introduction

Car manufacturers, for example, GM, DaimlerChrysler, commonly referred to as original equipment manufacturers (OEMs), are currently adopting flexible car architectures in their strategies to meet stiff competition by achieving higher levels of innovation in their products. A flexible car architecture allows, for example, decoupling functionality from underlying control, computing, and communication architectures thus simplifying the overall vehicle design [SAE06].

Developing a successful flexible car architecture is challenging due to the complexity of the subcomponents and also to the stringent requirements faced by the automotive manufacturers, for example, emissions, standards (e.g., CAFE), safety, comfort, etc. The challenge is particularly acute for the electronics, communications, and software subsystems and some techniques are being used to deal with this challenge such as model-based development (see Chapter 10) and the use of open standards (see Chapter 2). OEMs currently face stiff competition and they have to contend with the development of extremely complex electromechanical systems (e.g., a hybrid-electric vehicle [HEV]) in a relatively short development interval. The number of electronic control units (ECUs) and networks in modern vehicles continues to increase and there is a trend to use a backbone main network with several subnetworks as depicted in Figure 6.1. The subnetworks typically support each of the major automotive subsystems such as chassis, power train (engine and transmission), X-by-wire, body, and infotainment while the backbone network is used to support the entire vehicle communications. The requirements of the backbone and each of the subnetworks are not the same as detailed below.

6.1.1 Main Requirements of Automotive Networking

In general, from a technical standpoint, there are four main requirements for hierarchical networks such as those shown in Figure 6.1 for in-vehicle systems:

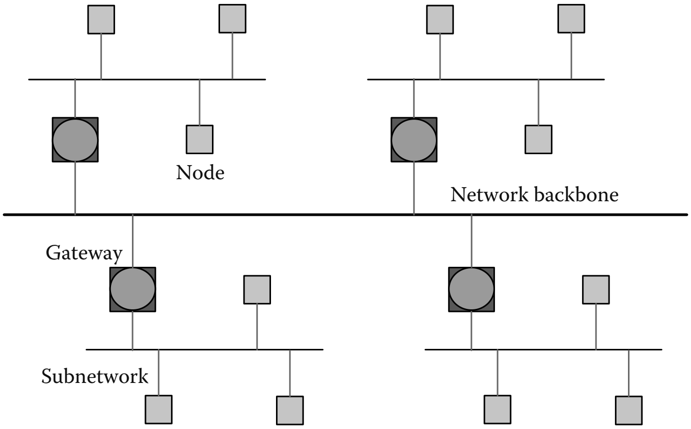


FIGURE 6.1 Network architecture of modern automobiles.

1. Deterministic behavior
2. High speed
3. Flexibility
4. Dependability

Flexibility is perhaps the most important requirement to be met by all networks, and indeed for all electromechanical vehicle components. The flexibility requirement is needed to achieve flexible car architectures. With the exception of flexibility, not all remaining requirements (i.e., determinism, high speed, and dependability) have the same level of importance or criticality depending on the type of network and the applications supported. For a backbone network, high speed is paramount whereas for a subnetwork that supports an X-by-wire subsystem, dependability is paramount. A brief explanation of these requirements follows.

6.1.1.1 Deterministic Behavior

A system exhibits a deterministic behavior when the performance measures of its services are predictable under a number of conditions and characterized by specific nonrandom equations. For in-vehicle networks, perhaps the most important performance measure is *message latency* defined as the time interval from when a transmitter node enqueues a message for transmission (i.e., a transmission request) until such message is successfully read (i.e., a reception) by a receiver node.

6.1.1.2 High Speed

This is mainly to support multimedia and Internet-related applications (e.g., infotainment) and inter subnetwork communications. High speed is relative, in the future up to 100 Mbps might be required for the backbone, to support global system integration including transfers of large amounts of information, and 1 Mbps for subnetworks, with millisecond range, few bytes long, monitoring and control transactions.

6.1.1.3 Flexibility

Flexibility is a requirement that has not been given the importance it deserves in most networking research and development endeavors. Its importance is only becoming apparent recently due to the high fierce competition among vehicle makers, the goal of achieving highly flexible car architectures, and the complexity of the underlying technologies. Flexibility is also difficult to define and characterize precisely due to its many contexts, meanings, and interpretations. One context is at the communication architecture level, another at the communication protocol level, and a third one at the communication system implementation level. Still another context is at the maintenance, repair, and service level. All contexts are important and when we discuss flexibility at the protocol level, we need to relate flexibility to all levels including at the architecture, implementation, and service levels. Thus, we need to take a holistic approach when dealing with flexibility.

6.1.1.4 Flexibility Attributes

Flexibility has a number of important attributes discussed below:

- *Design flexibility*—This is the flexibility of the communication architecture that enables making design choices in a simple fashion.
- *Configuration flexibility*—At the protocol level it means to put many choices and options in an easy way. At the system level it means using the network to configure the overall software including communication options.
- *Network load (traffic) flexibility*—Sometimes the offered network traffic changes, thus the communication architecture must be flexible in accommodating a wide variety of traffic patterns.
- *Reconfiguration flexibility (involving change)*—Redo a configuration to a vehicle after the vehicle has been in operation for a while.
- *Diagnostic flexibility*—When there is a problem with a vehicle, the communication capability supports diagnostics to figure out what is wrong with a vehicle's subcomponent.
- *Parameter flexibility*—The capability to monitor a wide set of system parameters.
- *Test flexibility*—The capability of performing a wide variety of tests in a simple and automated fashion.
- *Integration flexibility*—The capability of the communication architecture to easily support a wide variety of configurations at the system integration phase.
- *Hierarchical network flexibility*—The capability of the communication architecture to function in a hierarchical fashion as shown in Figure 6.1.
- *Functional flexibility*—The capability offered by the architecture to support a wide variety of vehicle functions.
- *Just-on-time flexibility*—The capability of the communication architecture to support any change, configuration, or reconfiguration in a very short time thus meeting tight deadlines on time.

6.1.1.5 Dependability

Dependability involves reliability, availability, maintainability, safety, integrity, and confidentiality [LAPR01]. Until now, only the first four attributes of dependability have been relevant for in-vehicle systems. However, integrity and confidentiality are now becoming more important since vehicles are becoming more and more networked with the external world, for example, car-to-car or car-to-road communications, Internet connection and integration in the car architecture or wireless vehicle access control and remote/wireless vehicle diagnostics. This openness of the vehicle architecture will require the use of security techniques and technologies, from firewalls to cryptography and identity certification, which have not been typically considered in the automotive domain but which are necessary to prevent unauthorized control over vehicle functions or unauthorized access to private data, for example, operational parameters from cars subsystems or even routes and usage logging. These security issues are, however, beyond the scope of this chapter.

With respect to the other attributes, in spite of the obvious importance of reliability and availability, the safety attribute is paramount. From a dependability point of view, applications can be safety-critical or non-safety-critical. The former are applications where a failure of a component or the entire system may lead to a loss of equipment, human injury, or loss of life. In current vehicles, the growing replacement of mechanical links by distributed electronic architectures, commonly referred to as X-by-wire systems, makes these architectures safety-critical. To prevent possible faults from disrupting steering, braking, accelerating, or causing engine failure, a correct fault hypothesis must be formulated and adequate fault-tolerance mechanisms must be considered and integrated in the architecture since design time. Such mechanisms can take advantage of a priori knowledge to distinguish correct from incorrect system states and this has been a strong motivation for the adoption of static designs that maximize the a priori knowledge available. Flexibility, on the other hand, tends to reduce such knowledge, thus leading to a conflict between flexibility and safety. Current vehicle requirements demand new ways to reconcile these aspects and improve flexibility without jeopardizing safety.

Finally, flexibility can also play in favor of dependability. For example, reconfiguration upon hazards with relocation of functionality from damaged nodes to operating ones is a consequence of flexibility and increases the system dependability by means of graceful degradation and survivability.

6.1.2 Networking Technologies

Small area fieldbus networks are well established in many application domains ranging from process control, manufacturing, medical, automotive, etc. [THOM98]. There are many established fieldbus communication protocols, but the most common include Profibus [TOVA99], Controller Area Network (CAN) [ETSC01], and the actuator sensor interface ASi network [MIRO99]. There are also specialized fieldbus protocols for highly dependable avionics and automotive applications such as TTP/C [TTPC], FlexRay [FLEX05], and SAFEBus [HOYM92]. Unlike other well-known network architectures such as the Internet, the common denominator of these so-called fieldbus networks is that they have a protocol stack consisting of just two or three layers as depicted in Figure 6.2. The CAN protocol is a two-layer fieldbus protocol originally developed by the Bosch Corporation in the early 1980s and intended for control applications [CAN91]. Currently, CAN enjoys widespread use in many application fields such as vehicles, home automation, railways, aerospace, marine, embedded machine control, robotics, factory automation, process automation, medical equipment, building automation, laboratory equipment, etc. [CAN04].

But is the best automotive networking solution already available or in progress? By best we mean networking technologies that meet the aforementioned requirements in a simple, effective, and low-cost fashion. Although it is too early to identify the best automotive networking solution at the backbone level, we believe that CAN is

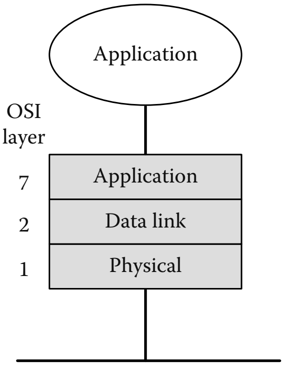


FIGURE 6.2 Communication architecture of a fieldbus device.

currently the best networking technology at the subnetwork level. CAN is not considered a high-speed protocol when compared to other alternatives, for example, FlexRay (see Chapter 5) and TTP/C (see Chapter 15). Nevertheless, CAN provides enough speed for in-vehicle monitoring, diagnostics, configuration, and control applications (see requirements above).

This chapter summarizes several proposals for CAN-based architectures suitable for in-vehicle subnetworks. All proposals aim at improving the network dependability in some way, while providing different levels of flexibility and determinism, exploiting to some degree the native CAN protocol. A truly flexible communication architecture will support flexible car architectures by simplifying product development, converting it to an integration task of chassis, power train, X-by-wire, body, and infotainment components.

6.1.3 CAN Features and Limitations

The CAN protocol has been around for about 20 years and has become important for many small area applications mainly due to its error control features, low latency, network-wide bus access priority, and instant bit monitoring [TIND95]. In addition to the aforementioned important features, CAN offers other excellent control features to recover from frame errors (including stuff bit errors, and CRC errors), which are not reviewed here as the reader is referred to the specification [CAN91].

Many important applications in automotive, robotics, process control, manufacturing, etc. are content with these features and this is evident from the large number of CAN-based applications worldwide. This is not to say there is no room for improvement. Application designers would be much happier if CAN could be made faster, cover longer distances, be more deterministic and more dependable [PROE00]. In fact, because of its limited dependability features, there is an ongoing debate on whether the CAN protocol, with proper enhancements, can support safety-critical applications [PIME04b,KOPE99].

The aim of this chapter is twofold: first, to characterize CAN limitations namely large and variable jitter, lack of clock synchronization, limited speed–distance product, flexibility limitations, data consistency issues, limited error containment, and limited support for fault tolerance; and second to discuss recent development and research being carried out to overcome these limitations [BARR06a,RUFI98, FERR05a,FERR06,PIME04b,FUHR00,BROS03a,RODR06].

6.1.3.1 Large and Variable Jitter

One of the most interesting features of CAN is its bandwidth-efficient arbitration mechanism, based on network-wide fixed priorities, which allows each node to try to transmit at any instant according to a carrier-sense multiple access (CSMA) protocol. However, this feature also has the negative side effect of causing a substantial network delay jitter because, without synchronization of the transmission instants, any node will encounter all possible interference patterns from higher priority traffic when trying to transmit. This jitter can be controlled, and sometimes eliminated, using global synchronization and relative offset adjustments.

6.1.3.2 Lack of a Clock Synchronization Service

As indicated above, the availability of a global synchronization can help to control the jitter. This synchronization and other important features of distributed embedded systems can rely on a clock synchronization service. Unfortunately, the CAN standard does not include such a service. Due to this, whenever a CAN-based distributed system requires a synchronized clock, it has to be provided at the application level. This is usually achieved by means of a software-implemented clock synchronization algorithm, although hardware implementations have also been proposed.

6.1.3.3 Limited Speed–Distance Product

This limitation is typical in shared serial data networks but it is particularly severe in CAN because of its dependency on instant bit monitoring while transmitting, a feature sometimes referred to as in-bit response. In CAN, the electrical signals must propagate over the whole network within a fraction of the bit time. Thus, the longer the network, the longer the bit time must be. Typical values are about 40 m network length at 1 Mbps and 1 km length for a transmission rate of 50 kbps. Overcoming this limitation while maintaining compatibility with the standard can be achieved, for example, with different topologies (e.g., star [BARR04]) or, possibly, with segmentation (e.g., using switches).

6.1.3.4 Flexibility Limitations

CAN is normally considered a highly flexible protocol in virtually all dimensions of flexibility referred above. However, the arbitration mechanism is based on the message identifiers that establish the message priority and must be unique across the system. The assignment of IDs to messages has, thus, a strong impact on the timeliness of the communications and forces a system-wide fixed priority message scheduling approach. If higher fairness is desired than achieved with fixed priorities scheduling, or to facilitate the assignment of IDs without strong consequences on the traffic timeliness, other mechanisms must be added to CAN, for example, dynamic update of IDs or external message scheduling by means of transmission control. Moreover, all kinds of flexibility that imply dynamic changes in the message set conflict with timeliness. Combining such flexibility with timeliness requires the addition of an admission control unit that verifies all submitted changes and rejects all those that would compromise timeliness.

6.1.3.5 Data Consistency Issues

Inconsistent communication scenarios are one of the strongest impairments to achieve high dependability over CAN. These scenarios occur due to specific protocol characteristics and they may reveal themselves both as inconsistent message omissions (IMOs), that is, some nodes receive a given message while others do not, and as inconsistent message duplicates (IMDs), that is, some nodes receive the same message several times while others receive only once. Inconsistent communication scenarios make distributed consensus in its different forms, for example, membership, clock synchronization, consistent commitment of configuration changes, or simply the consistent perception of asynchronous events, more difficult to attain.

6.1.3.6 Limited Error Containment

Despite its built-in error-containment mechanisms, based on error counters that can lead the network controller to bus-off state, CAN still presents several limitations in this matter. One of the limitations is that the built-in mechanisms are relatively slow to act, depending on the frequency and type of errors. Other limitation arises from the bus topology, as specified in the standard, since errors occurring in the node interfaces, bus lines, or its connections, spread freely through the network causing interference with correct traffic. This may also happen with replicated buses via common-mode failures. A possible solution to this limitation consists in segmenting the network, at the physical level, for example, using a star topology and point-to-point links. Finally, another limitation concerns the transmission of erroneous messages, in value or timing, despite correct framing. CAN includes no protection to contain the propagation of such errors. A typical fault of this kind in the time domain is the babbling-idiot fault in which a node remains transmitting a message more often than desired, strongly interfering with the rest of the traffic. Protection against this kind of faults may include specific hardware support, such as a bus-guardians, that is, a device attached to a node that verifies the respective transmissions, blocking untimely ones, as well as high-layer protocols that restrict the transmission instants of nodes.

6.1.3.7 Limited Support for Fault Tolerance

Safety-critical applications require very high levels of dependability (typically reliability). In order to reach these levels, fault-tolerance techniques must be used. Wide support for fault-tolerance functions is not a common feature in most fieldbus networks. CAN already provides advanced mechanisms that prevent some faults from causing a general system failure and even some specific CAN transceivers implement several mechanisms capable of tolerating specific permanent faults in the communication links. However, these mechanisms are not enough for safety-critical applications. Additional mechanisms are required in order to tolerate node failures and a permanent failure of the bus (i.e., to support node and bus replication).

The remainder of the chapter is organized as follows: In Section 6.2 we revisit in detail the specific topic of data consistency in CAN, given its importance and impact on dependability. Then, in Section 6.3, we present a set of techniques and protocols developed to provide improved dependability without jeopardizing flexibility, namely (Re)CANcentrate, CAN-Enhanced Layer (CANELy), flexible time-triggered CAN (FTT-CAN), and FlexCAN. These techniques/protocols cover the various layers of typical fieldbus networks, with (Re)CANcentrate operating mainly at the physical and datalink layers, CANELy focusing on datalink issues and partially on higher layer and FTT-CAN and FlexCAN being two higher layer protocols that may operate over commercial off-the-shelf (COTS) CAN controllers. Finally, the chapter includes a reference to a few other protocols that are somehow related to the topic of dependability and flexibility and then a conclusion is presented.

6.2 Data Consistency Issues

The specification of CAN [ISO93] claims that this protocol exhibits *data consistency* in the presence of transient channel faults (i.e., transient faults occurring at the transmission medium or at the transceivers). This means that within a CAN network it is theoretically guaranteed that a frame is either simultaneously accepted by all nodes or by none. Such property roughly corresponds to the *Atomic Broadcast* definition, and has led many authors to assume that CAN provides this service, which is of capital importance in many fault-tolerant and real-time distributed systems.

Nevertheless, it is well known that CAN does not always accomplish the pretended data consistency [RUFI98,PROE00,RODR03a]. In this section, after introducing the mechanisms that CAN incorporates to supposedly guarantee data consistency, we present some situations in which those mechanisms fail to provide this property, and discuss the likeliness of those situations. It is shown that even though some of the situations of data inconsistency are quite probable, some others only occur upon rare fault scenarios. This section also discusses some of the ways that have been proposed to overcome the inconsistency problems. Fortunately, the most common causes of data inconsistency can be easily avoided with a proper system design, which would fulfill the relatively low dependability requirements of most current CAN applications. However, some authors claim that if CAN is to be adopted in more critical applications then some extra mechanisms are required to guarantee data consistency even in those rare fault scenarios. Some of these mechanisms are introduced at the end of this section.

6.2.1 Management of Transient Channel Faults in CAN

In the presence of transient channel faults, CAN is often considered as providing the five properties listed below, which would correspond to the definition of Atomic Broadcast [RUFI98]:

- **AB1**—Validity: if a correct node broadcasts a message then the message is eventually delivered to a correct node.
- **AB2**—Agreement: if a message is delivered to a correct node, then the message is eventually delivered to all correct nodes.
- **AB3**—At-most-once delivery: any message delivered to a correct node is delivered at most once.
- **AB4**—Nontriviality: any message delivered to a correct node was broadcast by a node.
- **AB5**—Total order: any two messages delivered to any two correct nodes are delivered in the same order to both nodes.

In order to achieve data consistency, the CAN protocol defines some specific mechanisms for error detection and error signaling [ISO93]. These mechanisms strongly rely on a basic characteristic of CAN: the quasisimultaneous view of the bits throughout the network.

In a CAN network the length of the bit time is long enough so that all nodes are quasisimultaneously sampling the value of the same bit. At each instant, the bit transmitted through a CAN bus can take one of two values: *dominant* or *recessive*. In most implementations of CAN, the dominant value is represented by the logical "0" and the recessive value is represented by the logical "1." Only if all the nodes simultaneously transmit a recessive value, the resulting bus value will be recessive. In contrast, if any of the nodes transmits a dominant value, the bus value will be dominant.

As indicated above, the claimed data consistency in the CAN protocol is achieved thanks to its special error-detection and error-signaling mechanisms. CAN presents five error detection mechanisms that lead to five different kinds of errors, namely bit error, stuff error, CRC error, acknowledgment error, and form error. Furthermore, every CAN node keeps two error counters, called the transmission error counter (TEC) and the reception error counter (REC), which account for the number of errors the node has detected in the last transmissions and receptions, respectively.

Depending on the value of the TEC and the REC, the internal state of a CAN node may change. The initial state of any CAN node is called *error-active*. If one of the error counters reaches a given threshold then the node steps into the *error-passive* state, which means that many local channel errors have been detected but that the node can still participate in the communication in a degraded mode. However, if the error counters keep increasing and the TEC finally exceeds a second (and higher) threshold then the CAN node enters the *bus-off* state, in which it is not allowed to participate in the communication in any way.

Whenever a node in the error-active state detects an error thanks to the previously mentioned mechanisms, it signals this situation to the rest of nodes by sending what is called an *active error flag*. An active error flag consists of six consecutive dominant bits, and starts at least one bit after the error was detected. This flag will eventually violate a CAN protocol rule, for example, it can destroy the bit fields requiring fixed form and thus cause a form error. As a consequence, all the other nodes detect an error condition too and start transmission of an active error flag as well. After transmitting an active error flag, each node sends recessive bits and monitors the bus until it detects a recessive bit. Afterward, it starts transmitting seven more recessive bits. The eight recessive bit chain resulting on the bus is called *error delimiter*. This error delimiter together with the superposition of error flags from different nodes constitutes what is called an *error frame*. After the error frame transmission, the frame that was being sent is automatically rejected by all receivers and retransmitted by the original transmitter. This simple mechanism allows the *globalization* of local errors and provides tolerance to the transient fault causing the error. In this way, data consistency is supposedly achieved. Nevertheless, it is not always the case that local errors can be globalized.

6.2.2 Impairments to Data Consistency

Despite the special mechanisms that CAN incorporates for error detection and signaling, two impairments to data consistency have been reported. The first impairment is the presence of the error-passive state. According to the standard, a CAN node enters this state whenever either the TEC or the REC exceeds a given threshold. A CAN node being in the error-passive state would not use active error flags to signal channel errors. Instead, it would use an error flag made up of recessive bits (the so-called passive error flag), which in fact cannot always force the other nodes to see the error. Due to this, if this node is the only receiver observing the error then an inconsistency appears in the network because it will be the only receiver rejecting the frame.

The second impairment to data consistency appears in the error-active state, and is related to the special behavior upon error in the very last bit of the end of frame (EOF) field. Whenever the transmitter of the frame detects an error in this bit, it handles the situation as already explained: it starts the transmission of an active error flag in the next bit, it considers the frame transmission as being erroneous, and it retransmits the frame afterward. In contrast, if a receiver of the frame detects an error in the last bit of the EOF then it just accepts the frame as correct [RUFI98].

This special behavior implies that whenever an error in the last but one bit of the EOF is only detected by a subset of the nodes, the nodes belonging to this subset reject the frame and generate an error flag in the next bit, which is in fact the last bit of the EOF. Therefore, this error flag will not make the other nodes reject the frame. This would violate the property of data consistency, as there would be receivers rejecting the frame (those that detected the error in the last but one bit of the EOF) as well as receivers accepting the frame (those that detected the error flag generated by the former nodes) [RUFI98].

This inconsistency scenario would lead to one of two potential failures. If the transmitter detects the channel error and is able to retransmit the corrupted frame then those receivers that accepted the first frame will receive the same message twice. This failure is called an IMD. Conversely, if the transmitter does not detect the error or is not able to retransmit the corrupted frame then some of the nodes will never receive the message. This failure is called an IMO.

The error scenarios that may lead to IMD and IMO are thoroughly discussed in Refs. [RUFI98,PROE00,RODR03a]. From these papers it can be summarized that, in the error-active state, IMO failures are always caused by one of the following reasons:

1. A second channel error that makes it impossible for the transmitter to detect the error frame issued by the receivers that have detected the first error.
2. A failure (crash) of the transmitter that makes retransmission impossible.
3. A deliberate reduction of the time available for retransmission due to some system requirement. This is common in real-time variations of CAN that adopt techniques to enforce error containment in the time domain, such as TTCAN, TCAN, or FTT-CAN.

6.2.3 On the Probability of the Data Inconsistency Scenarios

Once it has been accepted that situations of data inconsistency exist in CAN, it is important to assess the likeliness of those situations, since the adoption of a costly fault-tolerance technique should be justified by the likeliness of the fault. This is particularly important for those fault scenarios leading to an IMO failure in the error-active state, because the solutions to tolerate them have higher cost in terms of both computation and communication overhead.

Regarding the inconsistency in the error-passive state, it is important to remark that having CAN nodes in the error-passive state is not a strange situation (yet it is not so frequent either). For instance, a node may enter this state because it has a faulty transceiver, which issues wrong values to the transmission medium, or because of an error burst caused by a strong electromagnetic interference. Therefore, this source of inconsistencies is a potential threat for dependable applications over CAN. In fact, some authors have calculated that the time spent by the nodes in the error-passive state may be very important for high values of the bit error rate, such as 10^{-3} [GAUJ05]. According to the same authors even the bus-off state can be reached too easily for high values of the bit error rate (e.g., in 40 s for a bit error rate of 10^{-3}) [GAUJ05].

Concerning the data inconsistency scenarios in the error-active state, there is little discussion about the likeliness of having IMD failures in CAN. For instance, in [ETSC01] system designers are recommended to bear in mind this possibility when designing a CAN system. In contrast, the probability of suffering IMO failures is still a controversial issue. There are four papers that have studied the probability of the error scenarios leading to IMO failures in CAN [RUFI98,PROE00,RODR03b, FERR05].

The paper by Rufino et al. [RUFI98] was the first one to report the scenarios of IMO failures. They also presented an analytical model that allows calculation of the number of inconsistent duplicates and inconsistent omissions per hour, although their analysis is incomplete as it only evaluated the probability of occurrence of IMO failures caused by a crash of the transmitter. They obtained results under the following conditions: a network of 1 Mbps, made up of 32 nodes, with an overall load of 90%, a frame length of 110 bits, assuming that the time required for the transmission of one frame from each node in the network is $\Delta t = 5$ ms, and that nodes may fail with a failure rate of $\lambda = 10^{-3}$ failures/h. Results were obtained under diverse bit error rate assumptions. For instance, assuming a bit error rate of 10^{-4}, they obtained values in the order of 10^{-6} IMO/h. Those values are larger than the reference value of 10^{-9} incidents/h, the safety number of the aerospace industry [POWE92], which is being adopted by the automotive industry as well [KOPE95,HAMM03].

The paper by Proenza and Miró-Julià [PROE00] reported new scenarios of data inconsistency. In particular, they reported the possibility of IMO failures caused by a second channel error affecting the transmitter. They also provided an analytical model to calculate the probability of these new inconsistency scenarios. Under the same conditions used for the evaluation in Ref. [RUFI98], they obtained values in the order of 10^{-3} IMO/h. Therefore, they showed that the new scenarios have probabilities not only greater than the reference value (10^{-9} incidents/h), but also greater than the previously reported scenarios.

The paper by Rodríguez-Navas and Proenza [RODR03a] reported the third cause of IMO and analyzed it in the context of TTCAN. In particular, they showed that a limitation of the available time for retransmission (in TTCAN frame retransmissions are always disabled) may significantly increase the probability of IMO failures. Taking the same network conditions of the two previous papers, they obtained values for TTCAN in the order of 10^3 IMO/h, which are too high so as to be neglected.

In contrast to these papers, some authors claim that the scenarios of message inconsistencies are actually very unlikely and do not represent a threat for the adoption of CAN in critical applications. In particular, recent research in the experimental assessment of the CAN bit error rate [FERR04,FERR05] has provided measures in different environments for the bit error rate that seem to show that the bit error rates used in Refs. [RUFI98,PROE00,RODR03a] were quite pessimistic. They claim that, according to their results, the number of IMOs per hour may be below the 10^{-9} reference number for a CAN network. However, despite being very useful as a first published attempt to measure the bit error rate in a real environment, the results of the experiments are not conclusive, and thus additional sources of experimental data should be made available.

6.2.4 Solutions to Really Achieve Data Consistency over CAN

The existence of the error-passive state is the first potential cause of data inconsistency in CAN. Many authors have proposed to avoid this state in order to improve the dependability of CAN-based systems [FERR98,HILM97,RUFI98]. This is easily achieved using a signal provided in many modern CAN circuits, called *error warning notification*. This signal is generated when any error counter reaches a certain value that is considered as an indication of a heavily disturbed bus (for instance, the value 96 in the Ref. [PHIL]). This is a good point to switch off the node before it goes into the error-passive state, thus ensuring that every node is either helping to achieve data consistency or disconnected.

Concerning inconsistencies in the error-active state, it is important to remark that IMD failures can be easily tolerated so they do not require adoption of any specific mechanism. As indicated in the CAN literature [ETSC01], duplicates can be tolerated with proper application design. For instance, messages that toggle a value should never be sent in a CAN network.

In contrast, solutions to tolerate IMOs in the error-active state are significantly more complex. There is one proposal [PROE00], which eliminates the possibility of an inconsistent frame validation at the lowest possible level of the system architecture: the CAN controller. Nevertheless, this consistency is achieved by slightly modifying the behavior of the error detection and error-signaling mechanisms at the end of the frame, and therefore this solution is not really compatible with standard CAN networks.

Apart from this nonstandard solution, there are various proposals intended to provide data consistency that are fully compatible with standard CAN networks. These solutions are mainly distributed protocols that rely on message exchanges. They however differ in the fault model addressed, and some of them in fact do not cover all of the potential causes of inconsistent omissions that have been mentioned above.

For instance, the solution proposed in Ref. [RUFI98] is based on a distributed agreement protocol, and only addresses the IMO caused by a crash of the transmitter. Specifically, they introduce three protocols: EDCAN, RELCAN, and TOTCAN. In EDCAN all the receivers retransmit the message after reception to overcome transmitter failures. This protocol satisfies all the Atomic Broadcast properties except Total Order, thus providing *Reliable Broadcast* [HADZ93]. In RELCAN the same properties are satisfied taking a more efficient approach. The transmitter sends a CONFIRM message after the successful transmission of the main message. Only in case the CONFIRM does not reach the receivers in a specified timeout they start the retransmission of the main message. Finally, TOTCAN satisfies all the Atomic Broadcast properties, including Total Order. Each time a receiver gets a duplicate of a message, it puts it at the tail of a queue. The transmitter sends an ACCEPT message after the successful transmission of the main message. When the receivers get the ACCEPT message, they fix the position of the message in the queue. In case the ACCEPT message does not reach the receivers in a specified timeout, they remove the corresponding message from the queue. A detailed description of these protocols can be found in Ref. [RUFI98].

In Ref. [LIVA99] a solution is proposed that relies on a specifically designed hardware circuit. This circuit is a dedicated circuit, called SHAdow Retransmitter (SHARE), which is intended to detect bit error patterns that may mean an inconsistent frame validation. Upon detection of these patterns, the SHARE circuit would retransmit the potentially corrupted frame, thus guaranteeing its reception even when the original transmitter is faulty. Afterward, Total Order is accomplished by means of an ordering scheme that the nodes implement. However, this solution only addresses IMO caused by a transmitter crash.

The solution suggested in Ref. [PINH03] also addresses only IMO caused by a transmitter crash, although it could be apparently extended to deal with IMO caused by a second channel error. This solution relies on the transmission of two additional messages, the CONFIRMATION and the ABORT messages, which helps the nodes to agree whether the transmission of a frame was consistent or not. This solution does not guarantee Reliable Broadcast if the transmitter crashes, only Agreement.

The protocol presented in Ref. [LIMA03] is a *consensus* protocol that allows the nodes of a CAN network to agree on a certain value despite the existence of inconsistent omissions. This solution tolerates IMO caused by either a transmitter crash or a second channel error.

It is important to highlight that it is not always clear how the mentioned solutions can be integrated with other protocols over CAN [RODR03b]. Much research has been already conducted in order to improve both the dependability and the real-time properties of CAN, so the current research focus is more on the integration of those solutions than on the provision of new ones. Some of the architectures described later on in this chapter are good examples of this integrated approach.

6.3 CANcentrate and ReCANcentrate: Star Topologies for CAN

The use of field buses in distributed control systems has been widely spread mainly due to their electrical robustness and low cost. One of the key causes of their low cost is the bus topology they rely on. However, the use of bus topologies implies some limitations regarding dependability. In a bus topology, components are attached to each other with scarce error-containment mechanisms. Thus, one single fault in any component (e.g., communication controller, transceiver, connector, wire, etc.) of a network that relies on a bus may generate errors that can propagate throughout the communication subsystem leading, in some cases, to a generalized failure of communication. Therefore, a bus topology presents multiple single points of failure.

In the particular case of CAN, some solutions have been proposed in the literature to increase error-containment capabilities: replicated buses [RUFI99,RUSH03], bus guardians [TIND95a,FERR05a], and the reconfigurable bus called RedCAN [FRED02]. However, due to the characteristics that are inherent to the bus topology, the former two techniques, even if they are used together, cannot prevent the existence of a multiplicity of single points of failure. For example, replicated transmission media may suffer from common-mode spatial proximity failures [STOE03] and nodes can still send incorrect information to all media, and a bus-guardian is useless for containing errors generated by a faulty medium and may also exhibit common-mode failures with the node it supervises. On the other hand, RedCAN [FRED02] does have the advantage of tolerating one fault in one bus segment and also has the potential to detect and isolate several kinds of node failures. However, it is still sensitive to network partitions upon a second fault and the diagnosis, location, and isolation of a faulty component, be it segment or node, require the execution of an algorithm in which all nodes must participate, thus increasing the complexity of the solution and the error detection and isolation latency. Finally, the multiple single points of failure that occur in a bus can only be prevented by RedCAN if each network adapter also performs detection of corrupted bit streams (bit-flipping errors). To achieve all these properties, RedCAN network adapters that must be installed in every node become substantially complex increasing nodes' probability of failure.

In contrast, star topologies may represent an effective solution to prevent the existence of multiple single points of failure. In a simplex star topology, each node is connected to a central element, the *hub*, by its own *link*. One advantage of a simplex star topology is that links only come into spatial proximity at the center of the star and, thus, the probability that different links suffer from common-mode failures is significantly reduced. In fact, the only chance for such kind of failure is a fault in the proximity of the hub. Also, network partitions are impossible with a star topology. But the most important advantage is that the center of the star, that is, the hub, can be designed to have a privileged view of the system, knowing the contribution from each node, bit by bit, through its corresponding link. In a CAN network this privileged view allows a hub to reach a capacity of error detection that cannot be achieved using a bus because in a CAN bus all nodes' contributions are irreversibly mixed. Hence, an adequate hub could enforce confinement of faulty transmission media and faulty nodes, by disconnecting the adequate hub ports. Furthermore, fault independence would be ensured between a guardian placed within the hub and the nodes this guardian would supervise.

Notice that the hub represents the unique single point of failure of a network relying on a simplex star topology. Thus, a simplex star can boost error containment when compared with a bus, which includes multiple single points of failure. Nevertheless, the existence of a single point of failure may represent an important drawback for safety-critical applications, which require a degree of reliability as high as possible. In order to eliminate such single point of failure a replicated star topology can be adopted. In a replicated star topology, more than one hub is used so that if a subset of hubs fail the nodes can still communicate through the remaining nonfaulty hubs.

Some of the potential advantages of simplex and replicated star topologies have been exploited by communication protocols such as TTP/C [BAUE02] and FlexRay [FLEX05]. TTP/C proposes a network with two star couplers that are provided with fault-treatment (fault diagnosis and fault passivation) mechanisms. These mechanisms deal with a fault model that mainly includes babbling-idiot, masquerading, and slightly-out-of-specification faults [KOPE03]. Similarly, FlexRay allows building multiple star topologies with or without redundant channels and, in addition, offers the possibility of combining star and bus topologies. FlexRay also enables the possibility of including within the star coupler a centralized guardian [FLEX05a] whose fault-treatment mechanisms are similar to those included in a TTP/C hub.

Some star topologies have also been proposed for CAN [CIA,RUCK94,IXXA05, CENA01,SAHA06,BARR06a,BARR05a]. Some of them are *passive stars* in the sense that the hub acts as a concentrator where all the incoming signals are coupled [CIA]. These stars present important disadvantages [BARR06a] concerning coupling losses, strong limitations on the star radius or in the bit rate, electrical problems, etc. Other types of stars are known as *active stars* [RUCK94,IXXA05,CENA01,SAHA06, BARR06a], which overcome some of the technical problems of passive stars. The active stars of Refs. [RUCK94,IXXA05,CENA01] rely on an active star coupler, which receives the incoming signals from the nodes bit by bit, implements a logical AND, and retransmits the result to all nodes. An alternative active hub proposed in Ref. [SAHA06] allows connecting either a node or a whole CAN bus to each of its ports. This hub includes a set of state machines that detect each dominant pulse received through a hub port and conveniently echo it to the other ports.

Unfortunately, these passive and active stars either do not address fault confinement, or only deal with a small set of possible faults. Moreover, some of them are not even compatible with the CAN protocol. A deeper discussion on the drawbacks of existing passive and active stars can be found in Ref. [BARR06a].

In contrast, two star topologies, called CANcentrate and ReCANcentrate (we will use the term (Re)CANcentrate when referring to both topologies at the same time), which have been proposed in Refs. [BARR06a,BARR05a], are specifically devised for error containment and fault tolerance. These two stars, which will be described in this section provide, for CAN features regarding dependability and flexibility that are similar to the ones offered by protocols such as TTP/C and FlexRay. Moreover, both stars are fully compatible with COTS CAN components, with CAN applications and CAN-based protocols, for example, CANopen, DeviceNet, FTT-CAN, or Flex-CAN. This compatibility also allows CANcentrate and ReCANcentrate to keep all the good dependability properties already provided by CAN [ISO93], for example, in-bit response, error-signaling mechanisms, etc.

6.3.1 Rationale

The main objective of both CANcentrate and ReCANcentrate is to boost reliability in CAN networks by means of fault treatment (fault diagnosis and fault passivation) and fault tolerance.

To better understand the objective of these stars in terms of fault treatment, the following concepts were introduced [BARR06a]: *severe failure of communication*, that is, when more than one node cannot communicate; and *point of severe failure*, that is, a point whose failure is *severe*, which comprises the common concept of single point of failure. When analyzing CAN buses, the following faults may lead to severe failures:

- Stuck-at-dominant and stuck-at-recessive faults, either in the nodes or medium, arising from, for example, short circuits to ground or battery, or malfunctioning or isolated controllers.
- Medium partition faults that occur whenever the network is physically broken into several subnetworks called *network partitions*.
- Bit-flipping faults that occur whenever a network component, either node or medium, exhibits a *fail uncontrolled* behavior, sending random erroneous bits with no restrictions in value or time domains.
- Babbling-idiot faults that occur whenever a node sends syntactically correct frames that are erroneous in the time domain, causing undesired interference.

CANcentrate and ReCANcentrate deal with faults that are related to the physical layer and that are independent of the application. Thus, they are able to confine the first three types of faults outlined above. No assumptions are made concerning the location, frequency, and duration of errors that may occur as a consequence of such faults. Additionally, a guardian could be included in the hubs of both stars to confine babbling-idiot faults. However, even if such a guardian is not included in the hub, notice that CANcentrate and ReCANcentrate are able to detect a babbling-idiot node if, from the communication subsystem point of view, the node manifests as being bit flipping. This may happen if a babbling node starts transmitting a CAN frame while another CAN frame is already being sent by a nonfaulty node, thereby corrupting it.

The hub of CANcentrate isolates any faulty network component, for example, cable, transceiver, etc., at the corresponding hub port, thereby preventing error propagation and thus the occurrence of a severe failure. Therefore, CANcentrate improves fault treatment in CAN by reducing the multiple points of severe failure exhibited by any other network based on a CAN bus to a unique single point of failure, that is, the hub. This has the relevant effect of decreasing the probability of severe failures in the communication system. This probability can be further decreased by reducing the probability of hub failure [BARR04].

In some applications, the degree of dependability achieved by CANcentrate could be not enough and the presence of a single point of failure unacceptable. In these cases, spatial redundancy at the hub level is required so as to tolerate permanent faults of the hub. ReCANcentrate provides this redundancy by using a replicated star topology that includes two or more hubs. Besides providing the same capacity of error containment as CANcentrate, ReCANcentrate further tolerates hub and link faults.

6.3.2 CANcentrate and ReCANcentrate Basics

In CANcentrate, each node is connected to the hub by means of a dedicated link that contains an uplink and a downlink (Figure 6.3). The hub receives each node contribution through the corresponding uplink, couples all the nonfaulty contributions and broadcasts the resulting coupled signal through the downlinks.

Figure 6.4 shows the internal hub hardware architecture [BARR06a]. It is constituted by three modules: the Coupler Module, the Input/Output Module, and the Fault-Treatment Module. The Coupler Module takes into account each port contribution ($B_{1,\ldots,n}$) that is nonfaulty; calculates the resultant coupled signal, B_0, by means of a logical AND; and broadcasts it to the nodes. The Input/Output Module is constituted by a set of transceivers. Each transceiver that is connected to an uplink translates the physical signal from that uplink into a logical form that can be understood by the other modules of the hub. Additionally, each transceiver connected to a downlink translates B_0 into a physical signal that is sent to the corresponding node through that downlink. Finally, the Fault-Treatment Module is constituted by a set of enabling/disabling units.

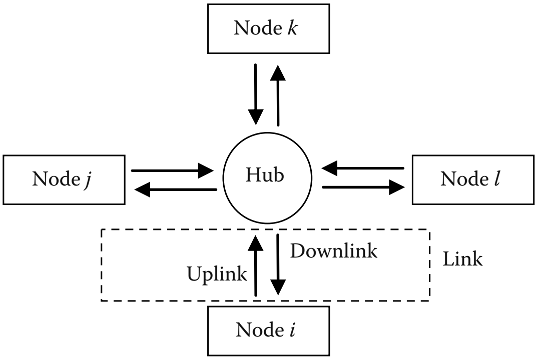


FIGURE 6.3 Connection schema of CANcentrate.

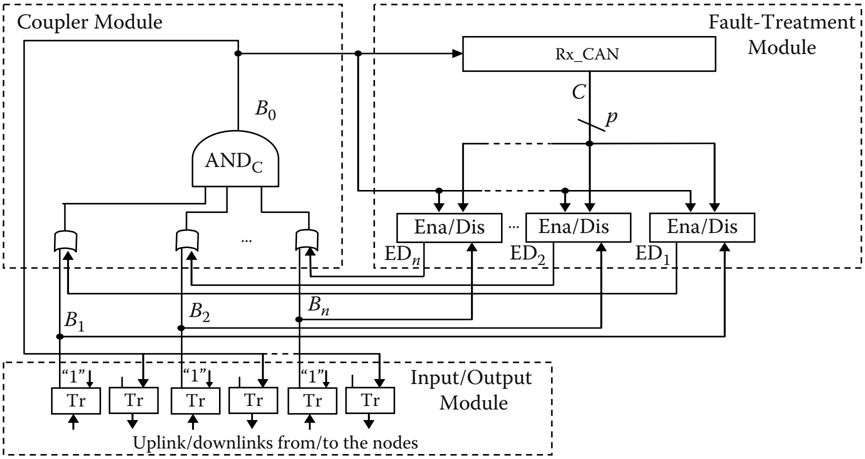


FIGURE 6.4 Internal structure of the CANcentrate hub.

Each one of these units monitors a given port contribution in order to detect errors. When a given port has accumulated too many errors, the coupled module isolates its contribution by driving a logical "1" through the corresponding enabling/disabling unit ($ED_{1,\dots,n}$) into an appropriate OR gate.

The use of an uplink and a downlink for each node allows separating the contribution of each node from the coupled signal, so that the enabling/disabling units can monitor each node contribution separately and detect faulty transmissions. This feature allows the hub to diagnose the location of faults with more precision than the typical error counters of CAN [ISO93]. Permanently faulty contributions are disabled, thus not propagated to the coupled signal, or in other words, confined to the port of origin. Additionally, for the sake of survivability, the enabling/disabling units implement a specific reintegration policy to reenable any port contribution after an error-free predefined time interval.

Moreover, since the hub carries out the coupling within a fraction of the bit time, its operation is transparent for the nodes. This makes CANcentrate fully compliant with CAN as referred before. However, a minor adaptation is still needed when connecting an ordinary CAN node to a CANcentrate port because of the separation between uplink and downlink. Using COTS transceivers, this connection requires two of them [BARR06a].

Although CANcentrate provides CAN with error-containment features that cannot be achieved by means of bus topologies, it is not tolerant to hub and link faults. However, ReCANcentrate [BARR05a] can provide such increased reliability by using a replicated star topology. Although ReCANcentrate does not limit the number of hubs, for the sake of simplicity, we consider only two hubs in the remainder of this section. The connection strategy is very similar to the one used in CANcentrate, with each node connected to each hub via an uplink and a downlink (Figure 6.5). The replication strategy is such that nodes transmit and receive the same data through all the stars in parallel and this is enforced transparently by a special coupling of the

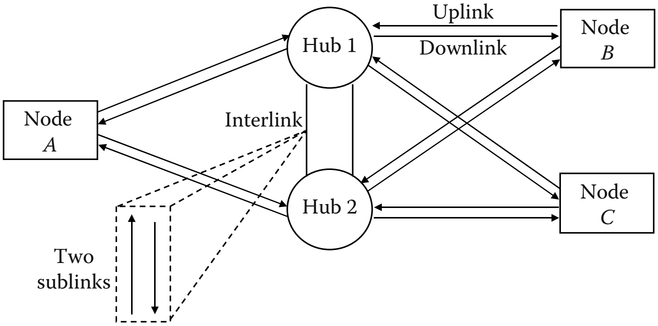


FIGURE 6.5 Connection schema of ReCANcentrate.

hubs using two or more dedicated links called *interlinks*, each containing two independent *sublinks*, one for each direction (Figure 6.5). Note that the use of more than one interlink allows tolerating interlink faults.

Internally, ReCANcentrate hubs are very similar to the ones of CANcentrate, but with some modifications mainly at the Coupler Module, using an AND coupling with two stages. In a first stage, each hub couples the contributions from its own nodes (i.e., the nodes that are directly connected to it), obtaining B_0, which now is called the *contribution of that hub*. The hub generates replicas of this contribution (B_{00} and B_{01} in Figure 6.6) and sends them to the other hub via one sublink of each interlink. In a

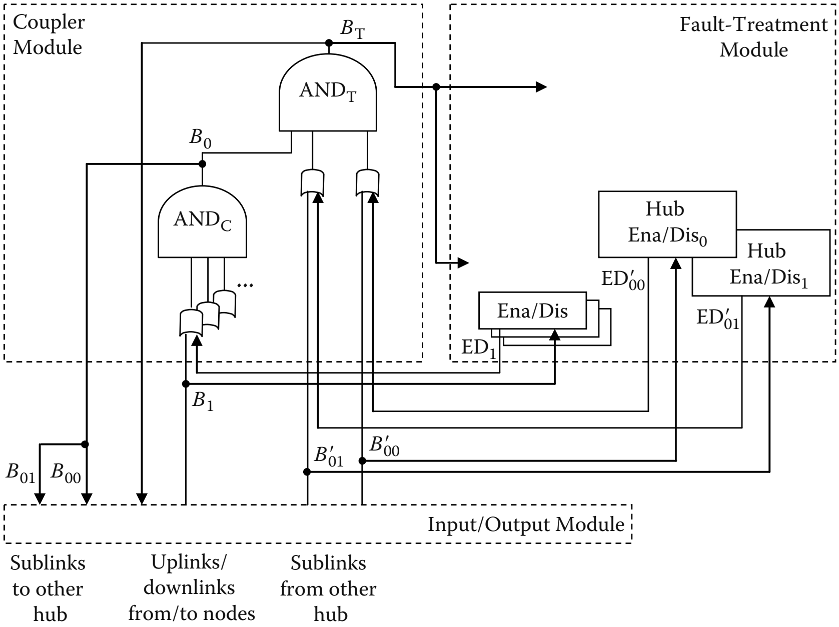


FIGURE 6.6 Two-stage AND inside ReCANcentrate.

second stage, each hub couples the replicas of the contribution of the other hub (B'_{00} and B'_{01}) with its own contribution, B_0, and then broadcasts the resulting signal, B_T, to the nodes that are directly attached to it.

This coupling scheme is necessary to allow each hub to monitor the other hub and isolate it in case of detecting errors. Such monitoring and isolation is carried out by specialized units, equivalent to those used for each port, called hub enabling/disabling units. In addition, when a hub fails it is also isolated by the nodes directly connected to it, for example, using an error count threshold with the error detection mechanisms included in CAN controllers.

The final coupled signal, which is broadcast to the nodes by all hubs, is unique and contains the contributions of all nodes in the system with at least one nonfaulty connection to one hub, no matter which. This enforces a consistent view of the network, that is, all connected nodes reach each other, even when some of them are connected to one hub, only. The possibility of connecting less critical nodes to only one hub allows reducing the cabling cost.

Beyond these advantages, ReCANcentrate is also fully compliant with CAN. The connection of a CAN node to ReCANcentrate still requires two COTS transceivers per link (per hub), as in CANcentrate. Moreover, a major advantage arises from the bit-level synchronization enforced by ReCANcentrate, which allows overcoming the difficult problem of replicated channel synchronization.

In case all interlinks fail, the above advantages of ReCANcentrate related to the bit-level hubs coupling are lost, but nodes can still communicate. For instance, all nodes could agree to use a unique hub for communicating. In this way, ReCANcentrate exhibits graceful degradation. Finally, ReCANcentrate also implements a reintegration policy similar to CANcentrate for long but nonpermanent node failures.

6.3.3 Other Considerations

Regarding the cabling, star topologies generally lead to longer cabling and higher costs than corresponding bus topologies, but not necessarily [BARR06]. In fact, the gains or losses in cabling length are highly dependent on the network's physical layout. Moreover, since star topologies yield substantial benefits of dependability when compared with bus topologies, star topologies should be the choice when dependability is an issue. Alternatively, (Re)CANcentrate allows combining both, bus and star topologies (Figure 6.7), connecting a set of nodes with lower fault-tolerance requirements to one hub port, sharing the same uplink and downlink. In such a way (Re)CANcentrate can provide hierarchical network flexibility.

Another aspect that deserves a reference is the product network length times bit rate, which is severely limited in CAN because of the in-bit response feature. However, this constraint is relaxed when moving from a bus to a star because in the former it applies to the total bus length while in the latter it applies to the star diameter, only, that is, the maximum summed length of any two links, which tends to be shorter. Nevertheless, the maximum diameter of a star is negatively influenced by the delays in the transceivers. Notice that in a bus, any communication must traverse two transceivers, four in CANcentrate, and six in ReCANcentrate. The design of specific high-speed transceivers for use within the hubs could reduce this problem.

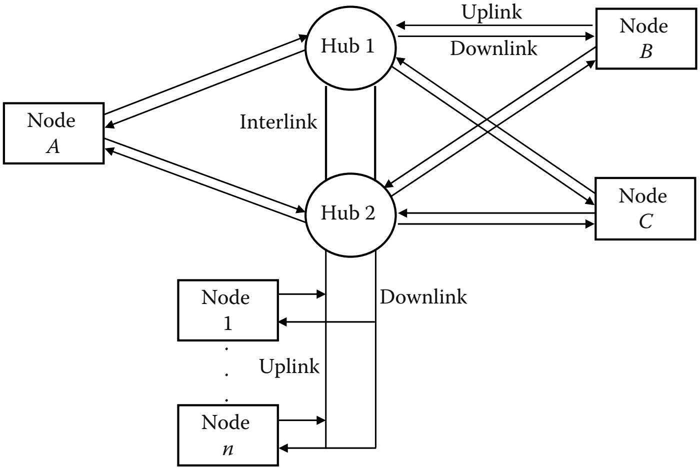


FIGURE 6.7 A replicated star topology with some nodes sharing an uplink and a downlink.

Finally, working prototypes of both CANcentrate and ReCANcentrate were developed using Field-Programmable Gate Array (FPGA) technology programmed with VHSIC Hardware Description Language plus the necessary interface components. The maximum star diameter achieved with CANcentrate at 690 kbps was of 70 m [BARR05], whereas for ReCANcentrate at 625 kbps it was of 25 m [BARR06]. This relation between the bit rate and the star diameter does not depend on the number of ports the hubs are provided with. In fact, the most influencing factor was the delay provoked by the commercial CAN transceivers located at the hubs.

With regard to the isolation and reintegration delays, at 625 kbps, the isolation delay of faults injected at hub link ports was 73 μs for stuck-at-dominant faults and ranged between 150 and 690 μs for bit-flipping faults. The isolation delay for the same types of faults at interlink ports was 216 μs and ranged between 476 and 2600 μs, respectively. Finally, the average time for two independently operating ReCANcentrate hubs, that is, without interlinks, to resynchronize upon interlinks connection and establish a single bitwise broadcast domain was of 1.3 ms [BARR06]. Notice that these latencies are quite short making (Re)CANcentrate adequate to demanding application domains such as in-vehicle networks, where transmission rates are typically below 1 kHz, that is, most of the referred faults can be detected and isolated in between two consecutive transmissions.

6.4 CANELy

The CANELy architecture was, to our best knowledge, the first attempt to overcome the dependability limitations of native CAN. More specifically, CANELy addresses the following limitations: lack of a clock synchronization service, data consistency problems, limited error containment, and limited support for fault tolerance.

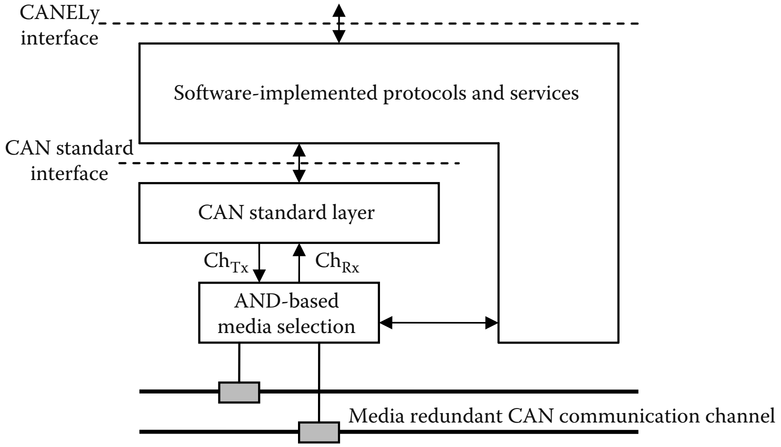

CANELy interface
Software-implemented protocols and services
CAN standard interface
CAN standard layer
Ch_{Tx}
Ch_{Rx}
AND-based media selection
Media redundant CAN communication channel

FIGURE 6.8 Basic structure of the CANELy architecture.

Figure 6.8 shows the architecture of CANELy. Note that the central component of the architecture is the standard CAN layer, which is implemented by one COTS CAN controller complemented/enhanced with some low-level protocols and simple machinery including a transparent medium redundancy scheme. In this way, the services of CANELy are mainly provided off-the-shelf, without modifications to the CAN standard or to available CAN controllers.

In the remainder of this section we briefly discuss how CANELy addresses the dependability limitations mentioned above. For a more detailed discussion, interested readers are referred to the literature in Refs. [RUFI98,RODR98,RUFI99,RUFI03, RUFI06].

6.4.1 Clock Synchronization

The CANELy architecture provides a fault-tolerant clock synchronization service [RODR98], which is also referred later in Section 6.7. This service is implemented in software, and achieves fault tolerance by means of an agreement protocol, which the nodes periodically execute in order to decide the right time reference in the system. This protocol may require an excessive number of messages for certain applications, but it tolerates a wide range of faults; including IMOs and Byzantine node failures, among others. The precision achieved is in the order of 100 μs.

6.4.2 Data Consistency

The problem of data consistency has received special attention in the CANELy architecture. In order to overcome this limitation, the solution proposed in CANELy is a suite of protocols (namely EDCAN, RELCAN, and TOTCAN), which guarantee different kinds of broadcast services [RUFI98]. The characteristics of these protocols were described previously, when discussing the data consistency problem in Section 6.2.

6.4.3 Error Containment

CANELy exhibits better error-containment properties than natural CAN, though it is a side effect of the media redundancy scheme of CANELy rather than an original goal of the architecture. In particular, CANELy includes some low-level mechanisms that may help detection of a faulty transmission medium or transceiver [RUFI99]. Nevertheless, the error-containment capabilities of CANELy still suffer the limitations inherent to the bus topology as pointed out in Section 6.3.

6.4.4 Support for Fault Tolerance

In contrast to standard CAN, CANELy provides a number of features that help the design of fault-tolerant applications. In particular, CANELy includes mechanisms to provide medium-fault tolerance and to support node replication.

Concerning medium-fault tolerance, the CANELy architecture uses a highly effective media redundancy scheme (the AND-based media selection block in Figure 6.8) for handling CAN physical partitions [RUFI99]. The received signals of each medium are combined in a conventional AND gate before interfacing the media access control layer. This secures resilience to medium partitions and stuck-at-recessive failures in the network cabling. Other cabling failures such as stuck-at-dominant can be also detected and treated with a short latency (209 μs). Beyond medium replication, CANELy incorporates another mechanism (the so-called inaccessibility control) to measure the network access delays as seen by one node and thus determine whether the network inaccessibility periods exceed certain prespecified tolerable limits [RUFI06].

Concerning node replication, CANELy includes two useful services: a service for reliable node failure detection and a group membership service [RUFI03]. The membership service is intended to provide consistent information about the state of a collection of *participants*, and can therefore be used to support upper layer services such as redundancy management, group communication, clock synchronization, etc. Both services are organized in a failure detection/membership protocol suite, which is built as a software layer on top of a CAN controller interface. In this way, upper layer protocol entities may request the local node to join/leave the set of active nodes or obtain the current node membership view. A notification of a change in the composition of the set of currently active nodes is due whenever a node joins/leaves the service or upon the detection of a node crash failure by the failure detection mechanisms.

Both the node failure detection and the membership services are based on signaling the node activity through the broadcast of a life-sign message similar to a *heartbeat*. In some cases, these life-sign messages can be piggybacked on periodic messages of the system, thus reducing the bandwidth required by the protocol. Due to space limitations, it is not possible to describe the mechanisms that CANELy specifies in order to consistently manage the life-sign messages as well as the join/leave messages [RUFI03]. These mechanisms are inspired on the CANELy broadcast protocols [RUFI99], which were reviewed in Section 6.2.

6.4.5 CANELy Limitations

CANELy addresses many of the limitations of CAN while maintaining, in general terms, the flexibility of original protocol. However, one common criticism is that some of the proposed services entail a substantial overhead in terms of message exchange, for example, the fault-tolerant clock synchronization. Therefore, the use of CANELy services requires a careful weighting of its suitability for each specific application. Error-containment impairments inherent to the bus topology, as referred above, are another source of limitations that subsists in CANELy. Moreover, the available literature has not provided indications on how to build a complete fault-tolerant system based on CANELy services. In fact, every mechanism has been proposed and described independently from the others. Nevertheless, CANELy can be seen as a set of useful mechanisms or techniques based on CAN COTS components that system designers may consider when designing a fault-tolerant CAN system.

6.5 FTT-CAN: Flexible Time-Triggered Communication on CAN

The growing importance of flexibility in the design of distributed embedded systems, and particularly automotive systems, has already been discussed in Section 6.1. Several of its dimensions refer to flexibility with respect to run-time operational aspects. We will thus refer to all such dimensions, generally, as *operational flexibility*. The quest for this type of flexibility has been motivated by a desire to support dynamic configuration changes such as those arising from hazardous events, evolving requirements, environmental changes, and online quality-of-service (QoS) management [BOUY05,LU02,SCHM01,BUTT02]. Generally, higher operational flexibility allows increasing the system survivability [SCHM01,SHEL04], for example, by supporting flexible modes and graceful degradation, as well as increasing efficiency in the use of system resources [BUTT02], particularly CPU and network bandwidth, carrying along an inherent potential to reduce system costs and improve its dependability.

For example, in order to provide fault tolerance, functional backup replicas must be available in different nodes. However, in many nonactive replication schemes they will be seldom used and thus, keeping their required resources permanently allocated, namely CPU and network bandwidth, will be inefficient. On the other hand, it would be more efficient releasing these resources for other active functions during normal operation, which thus could provide a better QoS, and requesting them back only when necessary, readjusting accordingly the resources allocated to the other functions, which would then revert to a lower QoS but still sufficient for safe operation. Even with active replication, upon failure, the number of replicas is reduced causing a redundancy attrition [BOND98]. This could be compensated for, activating dormant replicas in nodes that were used so far for other less critical operations. This flexible approach to fault tolerance may lead to substantial cost reductions with respect to static replication. However, this kind of flexibility is not compatible with static schedules, as acknowledged in Refs. [LU02,PRAS03], and it must be achieved under adequate control in order to keep the system in a safe state, even during adaptations.

The level of flexibility and safety referred above requires an adequate support from the computational and communications infrastructure so that relevant parameters of tasks and messages can be dynamically adjusted within specified bounds [ALME03]. However, performing this adjustment promptly in a distributed system is challenging because of the network-induced delays, the need to achieve a consensus among the nodes involved in the adjustment, and the need to enforce the adjustment synchronously in all nodes. Basically, the system architecture should meet the following requirements:

1. Bounded communication delays
2. Online changes to the processing/communication requirements with low and bounded latency, which requires dynamic task/traffic scheduling
3. Online admission control (to filter out change requests that would jeopardize system timeliness based on appropriate schedulability analysis) complemented with bandwidth management, with low and bounded latency
4. Sufficient resources for a predefined degraded but safe operating mode of all subsystems
5. Change attributes that define which online changes are permitted for each task/stream

The first requirement calls for an appropriate network access protocol that is deterministic and analyzable. The second and third requirements correspond to a dynamic planning-based traffic scheduling paradigm. The fourth and fifth requirements are ways of constraining flexibility to a set of permitted changes that always result in safe operational scenarios.

Meeting the above requirements motivated the development of the FTT communication paradigm, in which a global traffic scheduler and system synchronizer are placed in a central locus that also includes a database with all communication requirements and other system characteristics [ALME02,PEDR03,ALME03]. The fact that all communication and synchronization-related data are centralized in one node allows its prompt and efficient management. This node is called master and the scheduling of the system activities (tasks and messages) is periodically broadcast to the other nodes in a master/slave fashion using specific control messages. This architecture still follows the time-triggered model [KOPE97], according to which system transactions are triggered by time and not by external asynchronous events, being the required global notion of time enforced by the master, but allows complex online system updates with low latency as desired [ALME03]. To the best of our knowledge, the FTT paradigm has been the first attempt to introduce a high level of operational flexibility in the time-triggered communication model to support emerging applications that adapt to current environment operating conditions or that adjust the QoS dynamically. This paradigm has been applied to several networking technologies, leading to the protocols FTT-CAN [ALME02], FTT-Ethernet [PEDR02], and more recently FTT-SE [MARA06a], which are based on CAN, Ethernet, and microsegmented switched Ethernet, respectively. Practical applications of these protocols are reported in the literature concerning the control of mobile autonomous robots with

FTT-CAN [SILV05] and video surveillance with FTT-Ethernet [PEDR05]. This section focuses on FTT-CAN, covering the system architecture and its main components as well as the main efforts to provide fault-tolerant operation. FTT-CAN addresses CAN limitations by providing higher flexibility with respect to traffic scheduling and management in general, guaranteed timeliness with operational flexibility and online reconfiguration capabilities for resource-efficient fault tolerance.

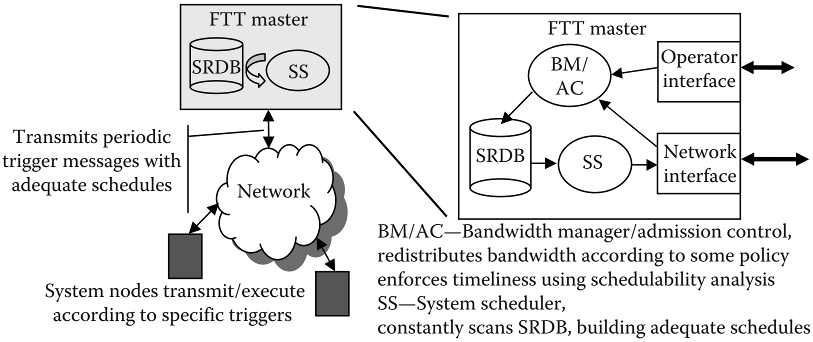
6.5.1 FTT System Architecture

The core of the FTT system architecture is the system requirements database (SRDB) that contains the current communication requirements plus other relevant system operational information (Figure 6.9). This component is located in a particular node called FTT master, together with the functional elements that operate over it, namely the system scheduler (SS), which builds the schedules that will be disseminated to the remaining nodes, and the bandwidth manager/admission controller, that manages the changes performed online in the SRDB and SS.

6.5.2 Dual-Phase Elementary Cycle

A key concept in the FTT protocols is the elementary cycle (EC), which is a fixed duration time slot used to allocate traffic on the bus. This concept is also used in other protocols such as Ethernet Powerlink and WorldFIP. The bus time is organized as an infinite succession of ECs. Within each EC there are two windows, each one dedicated to a different type of traffic, namely synchronous and asynchronous, which have time- and event-triggered characteristics, respectively. The order of these windows is protocol dependent and in FTT-CAN the asynchronous window precedes the synchronous one (Figure 6.10). The protocol enforces a strict temporal isolation between the two windows meaning that a transmission is only started if it finishes within the respective window.

The master node starts each EC by broadcasting a trigger message (TM). This control message synchronizes the network and conveys in its data field the identification


FTT master
SRDB
SS
Transmits periodic trigger messages with adequate schedules
Network
System nodes transmit/execute according to specific triggers
FTT master
BM/AC
Operator interface
SRDB
SS
Network interface

BM/AC—Bandwidth manager/admission control, redistributes bandwidth according to some policy enforces timeliness using schedulability analysis
SS—System scheduler,
constantly scans SRDB, building adequate schedules

FIGURE 6.9 System architecture of the FTT protocols.

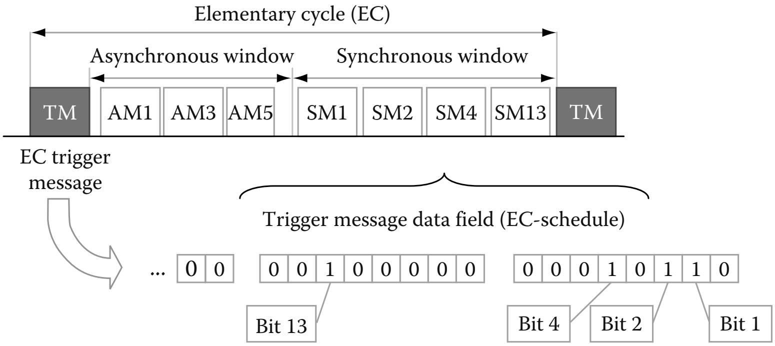


FIGURE 6.10 EC and EC-schedule encoding in FTT-CAN.

of the synchronous messages that must be transmitted by the slave nodes within the respective EC as well as specific trigger flags for task synchronization. This is referred to as EC schedule. All the nodes on the network decode the TM and transmit the scheduled messages in the synchronous window of that EC, with the collisions being sorted out by the native CAN arbitration mechanism. This kind of transmission control with just one control message issued per cycle is called master–multislave and saves substantial overhead with respect to common master–slave protocols because of both the reduction in control messages and the overlapping of turnaround times in all slaves.

The asynchronous traffic in FTT-CAN is also efficiently handled using the native arbitration of CAN. The protocol is designed in a way that the sequence of asynchronous windows behaves like a CAN bus operating at a lower bandwidth but still preserving its real-time properties. Asynchronous transmission requests issued outside those windows are transparently queued, possibly removed from the transmission buffers, and resubmitted in the following windows respecting the native arbitration process.

6.5.3 SRDB

The SRDB contains four components: the synchronous requirements table (SRT), the asynchronous requirements table (ART), the non-real-time table (NRT), and system configuration and status record (SCSR). The SRT includes the description of the current synchronous message streams:

$$\text{SRT} \equiv \{SM_i(C_i\, Ph_i\, P_i\, D_i Pr_i{}^{*}Xf_i),\ i = 1, \ldots, N_{\text{S}}\}$$

For each message stream, C is the respective maximum transmission time (including all overheads), Ph the relative phasing (i.e., the initial offset), P the period, D the deadline, and Pr a fixed priority defined by the application. Ph, P, and D are expressed as integer multiples of the EC duration. N_{S} is the number of elements in the SRT. $^{*}Xf_i$ is a pointer to a custom structure that can be defined to support specific parameters of a given QoS management policy, for example, admissible period values, elastic coefficient, value to the application, etc. It is also possible to define virtual messages, called *triggers*, which have $C = 0$ and are used to synchronize tasks executing in the nodes. These triggers are included in the EC schedule but do not cause any message transmission.

The asynchronous requirements component is formed by the reunion of two tables, the ART and the NRT. The ART contains the description of time constrained event-triggered message streams, for example, alarms or urgent reconfiguration requests:

$$\text{ART} \equiv \{AM_i(C_i \, \text{mit}_i \, D_i \, Pr_i), \, i = 1, \ldots, N_\text{A}\}$$

This table is similar to the SRT except for the use of the mit_i, minimum interarrival time, instead of the period, and the absence of an initial phase parameter, since there is no phase control between different asynchronous messages.

The NRT contains the information required to guarantee that non-real-time message transmissions fit within the asynchronous window, as required to enforce temporal isolation:

$$\text{NRT} \equiv \{NM_i(\text{SID}_i, \, \text{MAX\_}C_i, \, Pr_i), \, i = 1, \ldots, N_\text{N}\}$$

SID is the node identifier, MAX_C is the transmission time of the longest non-real-time message transmitted by that node, including all overheads, and Pr is the node non-real-time priority, used to allow an asymmetrical distribution of the bus bandwidth among nodes. N_N is the number of stations producing non-real-time messages. The master only needs to keep track of the length of the longest non-real-time message that is transmitted by each node.

The last component of the SRDB is the SCSR, which contains all system configuration data plus current traffic figures. This information is made available at the application layer so that it can be used either for profiling purposes or at run-time to support adaptive strategies, raise alarms, etc. It may occur that some of the SRDB components are not necessary in some simpler systems and in that case they are not implemented.

6.5.4 Main Temporal Parameters within the EC

One fundamental temporal parameter in the FTT protocols is the EC duration, which we consider to be E time units. This parameter establishes the temporal resolution of the system since all periods, deadlines, and relative phases of the synchronous messages are integer multiples of this interval. In other words, it sets the timescale of the synchronous traffic scheduler:

$$\forall_{i=1,\ldots,N\text{s}} Ph_i = k * E, \; P_i = l * E, \; D_i = m * E,$$
$$\text{with } k, \, l, \, m \text{ being positive integers}$$

Then, within the EC we can identify three intervals, the first of which is the time required to transmit the TM, which we consider constant and equal to LTM time units. It corresponds to an overhead that must be taken into account when scheduling.

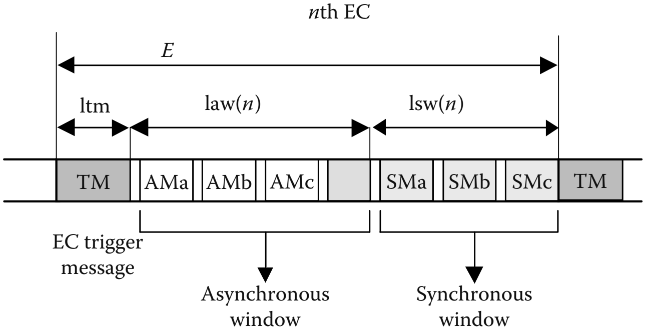


FIGURE 6.11 Main temporal parameters within the EC (FTT-CAN).

The following two intervals within the EC are the asynchronous and synchronous windows (Figure 6.11). The duration of this latter window in the *n*th EC, lsw(*n*), is set according to the synchronous traffic scheduled for it. This value is encoded in the respective TM, together with the EC schedule. The value of lsw(*n*) then determines the duration of the asynchronous window, law(*n*), and its end in the case of FTT-CAN. Basically, law(*n*) equals the remaining time between the TM and the synchronous window. The protocol allows establishing a maximum duration for the synchronous windows (LSW) and correspondingly a maximum bandwidth for that type of traffic. Hence, a minimum bandwidth can be guaranteed for the asynchronous traffic, too:

$$\forall_{n=1,\ldots,\infty} \quad 0 \le \mathrm{lsw}(n) \le \mathrm{LSW} \quad \text{and}$$
$$E - \mathrm{LTM} - \mathrm{LSW} \le (\mathrm{law}(n) = E - \mathrm{LTM} - \mathrm{lsw}(n))$$

Finally, the strict temporal isolation between both types of traffic is enforced by preventing the start of any transmission that would not finish within the respective window. In the synchronous phase this is enforced by the traffic scheduler so that all the messages specified in the EC-schedule always fit within the maximum width, LSW. In the asynchronous window of FTT-CAN this isolation is enforced by setting a timer to expire before the end of the window by an interval of time corresponding to the maximum transmission time. Requests that are pending when the timer expires may not be served up to completion within that interval and thus are removed from the network controller transmission buffer. This mechanism may lead to the insertion of a short amount of idle-time at the end of the asynchronous window (α in Figure 6.11), which must be considered when analyzing this traffic. Another amount of idle-time could have been inserted by the synchronous scheduler in a given EC to guarantee that the synchronous traffic does not extend beyond the end of the synchronous window. However, that amount of time is not considered in the value of lsw(*n*), just the effective transmissions, thus being implicitly reclaimed for the asynchronous traffic. In any case, it must be equally considered when analyzing the traffic timeliness.

6.5.5 Fault-Tolerance Features

There are two single points of failure in the FTT architecture, that is, the master and the bus channel. In fact, if the master node fails, no more TMs with EC schedules are transmitted and thus communication is disrupted. Network partitions also disrupt communication and thus must also be tolerated. This is prevented using replication, with one or more similar nodes acting as backup masters and two or more bus channels. Moreover, the nodes are assumed to be fail-silence, which is enforced in the slave nodes, in the time domain, with bus guardians and in the masters, in the time and value domains, with a particular network interface that supports internal node replication [FERR05]. Fail-silence in the value domain for the slave nodes is left for the application, if necessary, given its high cost. The reference fault-tolerant architecture is shown in Figure 6.12.

The master replicas have a further requirement of being synchronized so that they can replace the active one without any discontinuity of the traffic schedule. This requirement is particularly difficult with online system adaptation, because the communication requirements can evolve over time. Therefore, several mechanisms were developed to handle the master replication, namely

1. Master replacement scheme
2. Policing mechanism to detect loss of synchronization in backup masters
3. Protocol to synchronize starting or restarting backup masters consisting on transferring the current SRT
4. Protocol that ensures consistent SRT updates upon change requests

All the mechanisms related with master replication and fail-silence enforcement are described in Ref. [FERR06] and, in more detail, in Ref. [FERR05]. The former ones were further evaluated experimentally in Ref. [MARA06].

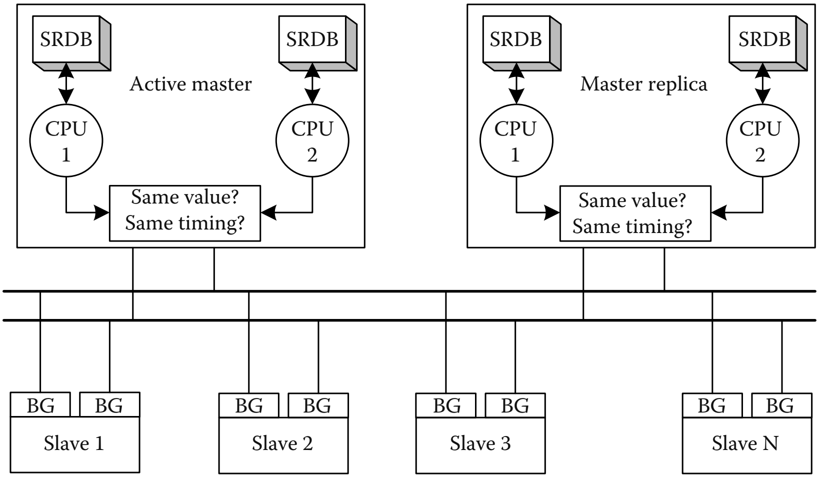


FIGURE 6.12 Fault-tolerant FTT-CAN reference architecture.

Finally, bus replication in FTT-CAN has been recently addressed in Ref. [SILV06] being proposed for duplicated transmission of critical messages as well as for transmission of different noncritical messages in each media, thus increasing the throughput with respect to a simplex bus. Another possibility is using a transparent bus replication mechanism such as the one proposed in Ref. [RUFI99]. An alternative to bus redundancy would be the use of a replicated star such as ReCANcentrate, also described in this chapter.

6.5.6 Accessing the Communication Services

The access to the communication services in the FTT protocols is carried out by means of two subsystems, the synchronous communication system (SMS) and the asynchronous communication services (AMS). The former follows the producer–consumer model [THOM99] to handle the synchronous traffic with autonomous control, that is, the network determines the transmission instants. The data are passed between application and network by means of shared buffers. The SMS_produce service allows writing a message in the appropriate buffer in the network interface; and the SMS_consume service allows reading from a message buffer in the network interface. For both services there is an option to synchronize with the network traffic, which allows triggering application tasks synchronously with the global schedule produced by the master. Three additional services allow managing the SRT, that is, SRT_add, SRT_remove, and SRT_change, which automatically invoke an online admission control to assure continued timeliness.

The AMS provides event-triggered communication services that, in the case of FTT-CAN, map directly onto the underlying CAN protocol. These services are AMS_send for nonblocking immediate transmission, and AMS_receive for blocking reception. More complex and reliable exchanges, for example, requiring acknowledge or requesting data, must be implemented at the application level, using the two basic services referred above. The AMS use queues whose length must be properly set at configuration time.

6.6 FlexCAN: A Deterministic, Flexible, and Dependable Architecture for Automotive Networks

In this section we summarize FlexCAN, one of the various architectures that has been proposed to overcome CAN limitations with respect to determinism and dependability. As explained below, FlexCAN incorporates several mechanisms similar to TTCAN, FTT-CAN, and FlexRay.

6.6.1 Control System Transactions

Nearly all embedded systems involve control systems of one kind or another. Basically, a control system takes input (I) measurements from a physical process (P), performs a control function (C), and produces outputs (O) to effect changes on the physical process. From a software perspective, the fundamental software functions or tasks found within control systems constitute a control system transaction (CST) [PIME06a]. A CST simply consists in performing a series of software tasks involving task (I) for collecting inputs from a process, task (C) to implement a control function, and task (O) to send controller outputs back to the physical process. These software tasks have to be performed in the following precise order:

$$\mathrm{P} \rightarrow \mathrm{I} \rightarrow \mathrm{C} \rightarrow \mathrm{O} \rightarrow \mathrm{P}$$

which is known as the CST precedence constraint. Together, the CST precedence constraint and the timing requirements of a specific application constitute the CST *timing*. The CST precedence constraint has two important properties:

Causal property (CP): The arrow implies a cause–effect relationship (e.g., P effects I).

Timing property (TP): The arrow implies a timing relationship (e.g., P → I happens before I → C).

Distributed functions

In a distributed embedded system the various sensors, actuators, and software processes I, C, and O are implemented in different ECUs. To convert a centralized CST (involving just a single ECU) into a distributed CST (involving more than one ECU) the system must be partitioned.

The partitioning points are depicted in the following more detailed signal flow:

$$\mathrm{P} \rightarrow \mathrm{S} \rightarrow \mathrm{I} \rightarrow /_{\mathrm{P}} \rightarrow \mathrm{C} \rightarrow /_{\mathrm{P}} \rightarrow \mathrm{O} \rightarrow /_{\mathrm{P}} \rightarrow \mathrm{A} \rightarrow \mathrm{P}$$

where the symbol $/_{\mathrm{P}}$ means a partition point. Thus, there are three partition points of interest, a partition point between the input and control software processes (p_{IC}), between the control and output processes, (p_{CO}), and between the output and actuator software processes (p_{OA}).*

Every time there is a partition point, a communication system is involved and from a software perspective, two additional software tasks are introduced to handle communications, a *transmitter task*, Tx, and a *receiver task*, Rx. Thus, the most general signal flow is

$$\mathrm{P} \rightarrow \mathrm{S} \rightarrow \mathrm{I} \rightarrow \mathrm{Tx} \rightarrow \mathrm{Rx} \rightarrow \mathrm{C} \rightarrow \mathrm{Tx} \rightarrow \mathrm{Rx} \rightarrow \mathrm{O} \rightarrow \mathrm{Tx} \rightarrow \mathrm{Rx} \rightarrow \mathrm{A} \rightarrow \mathrm{P}$$

It is the job of a communication system to provide a communication hardware and software architecture and make available Tx and Rx to end applications. FlexCAN is a communication architecture that provides such Tx and Rx synchronized to a time-triggered communication cycle enabling the enforcement of CST precedence constraints. There are also additional deterministic, flexible, and dependable features as explained next.

* It is not possible to have partition points between P and S and between A and P because these interfaces are hardware. Although it is possible to have a partition point between S and I, it is outside the scope of FlexCAN.

6.6.2 FlexCAN Architecture

The goal of the FlexCAN architecture is to provide additional deterministic and dependable capabilities to the CAN protocol without compromising its flexibility. This is accomplished by incorporating an additional layer on top of CAN. The main features of the FlexCAN architecture are replicated components, support for time–domain composability, replica synchronization, replication management, and enforcement of fail-silent behavior.

As depicted in Figure 6.13, the FlexCAN architecture can incorporate several replicated channels and several replicated nodes (in addition to normally nonreplicated ones) in a flexible fashion [PIME04b]. A node and all of its replicas are called fault-tolerant units (FTUs), but not all nodes in a network need to be replicated. Time–domain composability has been a major liability of CAN-based networks for safety-critical applications. There have been several proposals to overcome this limitation such as TTCAN [FUHR00] (Section 6.7.1) and FTT-CAN [ALME02] (Section 6.5), which basically adopt a time-triggered transmission scheme at a high level and still use CAN at the lower level. FlexCAN also adopts the time-triggered paradigm by using reference messages. The interval between reference messages is called the *communication cycle*, and this interval is further divided into a number of *subcycles* as shown in Figure 6.14. Whereas TTCAN uses a clock synchronization mechanism to implement the time-triggered scheme, FTT-CAN uses a master node to generate the reference messages. In FlexCAN, node-replication not only handles node failures but also supports dependable reference messages on a distributed basis without the need of synchronized clocks or master–slave schemes. The communication cycles, together with their subcycles, not only help with time–domain composability but also help to synchronize replicated nodes and channels, and to enforce fail-silent behavior, particularly using bus guardians. Furthermore, communication cycles are

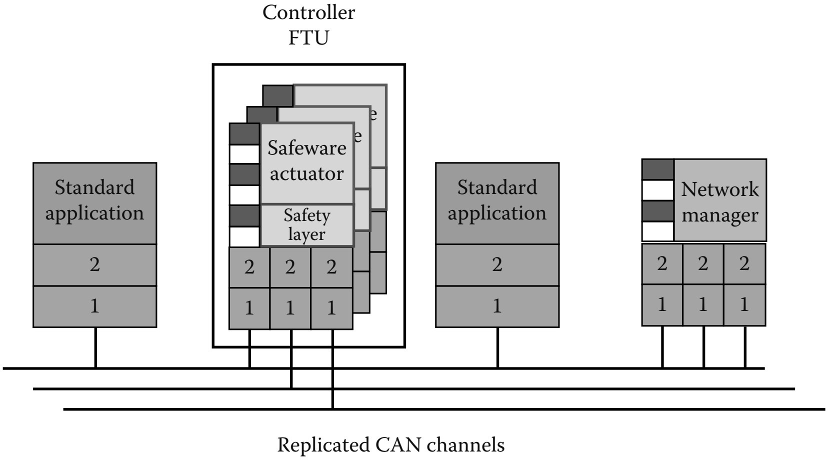


FIGURE 6.13 The FlexCAN architecture.

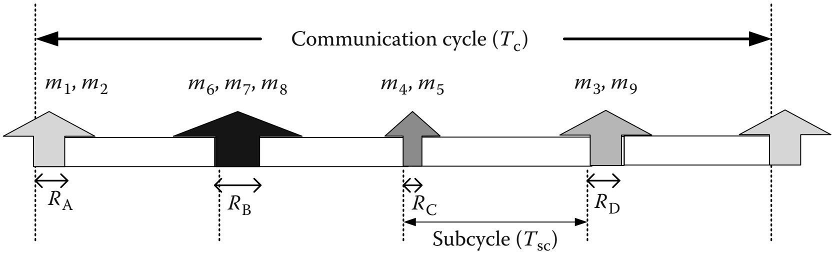


FIGURE 6.14 Time-triggered structure of the FlexCAN architecture.

useful to synchronize applications. FlexCAN is flexible in that it supports both periodic and aperiodic traffic. Multirate periodic messages (i.e., periodic messages with different periods) are also handled through appropriate message scheduling.

Providing message synchronization on replicated channels in FlexCAN is simple, as each node simply sends the same message on all replicated channels in an atomic fashion (i.e., without interruption). However providing message synchronization on replicated nodes is not trivial, and thus a special protocol known as SafeCAN is provided to handle node replication management [PIME04a,PIME04]. There are a number of fault-tolerant features in SafeCAN. The replacement of the primary node (called the next primary node) is always ready (provided the hardware is available). In terms of the type of redundancy algorithm used, FlexCAN uses a combination of static and dynamic redundancy. The SafeCAN protocol assumes that nodes are fail silent. To enforce such a fault model, FlexCAN uses a similar technique proposed in the FTT-CAN protocol to remove the message from its transmit buffer after a certain interval called the transmission attempt window (TAW) and also by using a special purpose bus guardian [BUJA05,BUJA06].

In the following, additional details of the time-triggered feature are given. The basic cycle T_c is divided into Q subcycles each of equal length T_{sc}. Regardless of their node location there can be up to P messages allocated per subcycle. Thus the maximum number of messages in the network is $P \times Q$. The goal is to have all messages in a subcycle transmitted before the next subcycle. This is based on a principle of time independence. Clearly, if this is enforced there will be no message queuing from one subcycle to the next and therefore from cycle to cycle. This peculiar message allocation scheme is the one adopted by FlexCAN. All message allocations are done in an offline fashion, just like TTCAN. Unlike TTCAN, this scheme does not require clock synchronization across all nodes, but simply requires management of timers with a minimum resolution of about 0.1 ms.

Just like TTCAN, to implement the time-triggered scheme, FlexCAN requires a synchronization message to explicitly mark the beginning of each cycle. But unlike TTCAN, there is no need for node clocks to be synchronized. Instead, FlexCAN relies on timers to divide the entire cycle into Q (e.g., four) subcycles. TTCAN requires clocks to be synchronized because the exclusive windows are allocated to a single message. On the other hand, in FlexCAN the subcycles are allocated to a group of messages, thus requiring less precision in the definition of the beginning and end of the time windows.

Time synchronization between a TT global message schedule and end applications is crucial for most control systems. FlexCAN enables the synchronization of any CST event (E_1 through E_8 in Figure 6.15) to the communication cycle. Figure 6.15 depicts the synchronization of message m_1 with the beginning of the communication cycle while other messages are synchronized with the beginning of subcycles (e.g., m_3, m_4, or m_6).

The main advantage of FlexCAN over TTCAN in terms of time-triggered and dependable features is that TTCAN has disallowed frame retransmissions, a notable feature of CAN for dependable operation, whereas FlexCAN lets CAN retransmit a frame in error but up to a certain time limit. Leaving enough time in each subcycle for error retransmission is important for dependable applications. Thus FlexCAN uses CAN fault-tolerant properties and enforces strict message deadlines. FlexCAN tolerates the following kinds of faults: transient arbitrary faults, permanent hardware faults, and permanent software babbling-idiot faults [BUJA05,BUJA06]. The semantics of these faults are as follows. Transient arbitrary faults are the kind of faults that are detected by the native CAN protocol and result in error and/or overload frames. Permanent hardware faults are permanent faults in the communication controller, transceiver, or bus, and they are masked by redundant nodes or channels. Permanent

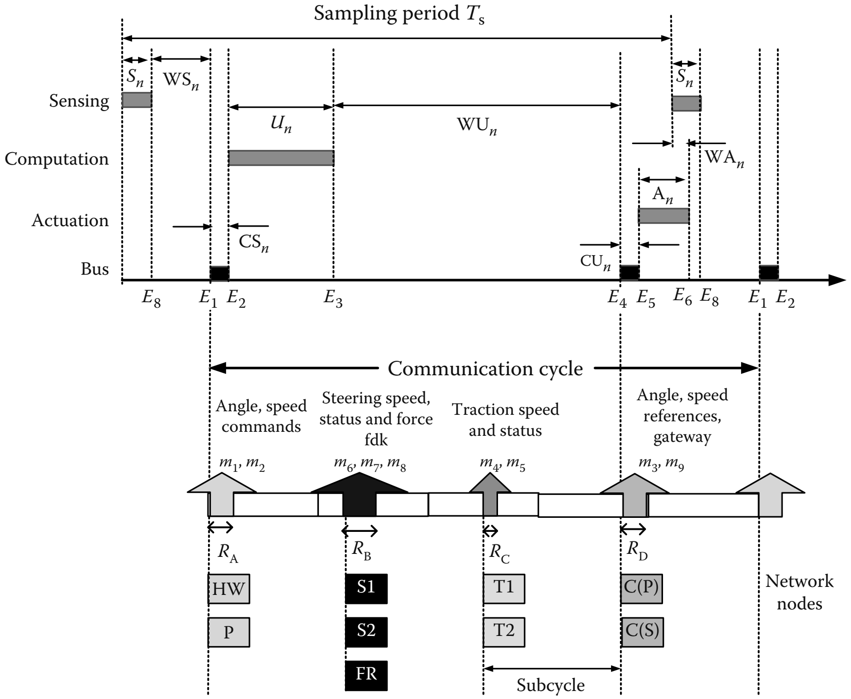


FIGURE 6.15 FlexCAN global message schedule synchronized to a CST functional unit.

software babbling-idiot faults are caused by software errors in a host controller that uses the bus with wrong values and at wrong times.

6.6.3 How FlexCAN Addresses CAN Limitations

In the following we detail how FlexCAN addresses the CAN limitations identified at the beginning of this chapter.

6.6.3.1 Deterministic Behavior

It is easy to design and verify message deterministic properties on a system based on a TDMA system such as FlexRay because messages occupy specific slots in the communication cycle. It is also possible to design and verify deterministic properties on a CAN-based system although the process is more complex because of the following difficulties:

1. Lack of a time reference in the CAN protocol
2. Widely different message transmission rates (or periods) that are possible

One difficulty with CAN latency calculations stems from the usage of CAN in early applications where a CAN network was the only network in the system and thus had to support all messages with widely varying message transmission periods. In fact, the application used in Ref. [BROS04] has six messages with the smallest and largest period of 2 and 240 ms, respectively, a factor of 120. With such widely differing factors in the transmission periods, the number of times a higher priority message is sent in an interval of duration t (i.e., the interference from higher priority messages) is not known ahead of time and must be calculated. This leads to a convoluted set of recurrent equations that must be solved recursively [TIND95]. Current and future systems have several communication networks each supporting an application type (e.g., entertainment, dashboard, power train, steer-by-wire, etc.). These networks are typically configured as a backbone network and several subnetworks each dedicated to one or few functional units of a vehicle. Messages in the subnetworks do not have widely varying message transmission periods, in fact, factors of 4–20 are sufficient. Another difficulty stems from the event-triggered nature of the CAN protocol that does not use the notion of global time. Because of this, there is uncertainty in the assumptions of values for the queuing jitter and the interference due to lower priority messages.

FlexCAN overcomes these previous two limitations by defining a subnetwork for messages with closely related transmission periods and also by defining a time-triggered time reference made of communication cycles divided into a number of subcycles. Just as is the case with other TT architectures, FlexCAN requires an off-line global message schedule to be configured. Furthermore, all messages scheduled in a subcycle are submitted for transmission at exactly the same time (at the beginning of the subcycle). For example, Table 6.1 shows a communication cycle with four subcycles where the rates (reciprocal of communication period) of message m_1 and m_2 are four and two times, respectively, the rate of the remaining messages.

TABLE 6.1 Example of FlexCAN Message Scheduling Supporting Multirates Using Four Subcycles

Subcycle	I	II	III	IV
Messages	m_1, m_5, m_6	m_1, m_2, m_3	m_1, m_4	m_1, m_2, m_7

The above described features greatly simplify the equations for calculating message latencies in the FlexCAN architecture [PIME06a]. To begin with, message latency calculations are done on a subcycle basis for all subcycles. For each subcycle, the messages are relabeled as m_1, m_2, etc. with m_1 the highest priority message, m_2 the next highest priority message, and so on. Because all messages in each subcycle are submitted for transmission at exactly the same time, the jitter is zero and there is no blocking from lower priority messages, resulting in simplifications in the calculation of message latencies.

Calculating message response times in FlexCAN has the following advantages when compared to that in CAN:

1. The calculations are more accurate since they are done on a subcycle basis relative to a time-triggered communication cycle with each subcycle being independent from the next (i.e., all messages are sent in their respective subcycles).
2. The calculations are easy to evaluate as the equations do not involve a recurrent relation.
3. The calculations assume that the jitter is zero, which is enforced by the FlexCAN message schedule.
4. The formulae are accurate because there is no need to consider the blocking time due to lower priority messages.
5. The multirate case is taken into account ahead of time by the FlexCAN message global schedule.

6.6.3.2 High Speed

Since many automotive applications at the subnetwork level do not need high speed, this inherent CAN limitation is not addressed by FlexCAN. The high-speed requirement is most prevalent for in-vehicle backbone networks.

6.6.3.3 Flexibility

As noted, FlexCAN relies on a static schedule for periodic traffic with enough bandwidth reserved for aperiodic traffic. Thus all flexibility attributes of FlexCAN assumes this context. The FlexCAN architecture is highly flexible providing the following types of flexibility.

6.6.3.3.1 Design Flexibility

This includes the support of subnetworks with different data rates and the support of flexible car architectures.

6.6.3.3.2 Configuration Flexibility

All deterministic and dependable features of FlexCAN are optional. A FlexCAN architecture without any additional features is simply CAN. As one or more of the following additional features are used, one can design a highly deterministic system (incorporating features 1 and 2) or a highly dependable system (incorporating all of the features) or anywhere between these two extremes.

1. TT-synchronization
2. Fail-silence enforcement
3. Channel replication
4. Node replication
5. Bus guardians

6.6.3.3.3 Network Load (Traffic) Flexibility

Support of periodic and aperiodic traffic, flexible message scheduling, support of multirate messages. Aperiodic traffic is supported by leaving enough bandwidth in each subcycle for this type of traffic.

In addition, FlexCAN does not compromise the flexibility inherent in CAN involving the following CAN flexibility features: reconfiguration flexibility (involving change), diagnostic flexibility, parameter flexibility, test flexibility, integration flexibility, hierarchical network flexibility, functional flexibility, and just-on-time flexibility. However, it must be noted that the aforementioned features are off-line rather than online.

6.6.3.4 Dependability

As noted, FlexCAN includes a number of optional features to support safety-critical applications. The features are fail-silence enforcement, channel replication, node replication, and bus guardians [BUJA05,BUJA06]. All of these features have been published in several papers [PIME04a,PIME04a,PIME04,BUJA05,BUJA06] and these features have been extensively tested [PIME06,BERT06]. Because of space limitations, how FlexCAN implements these features is not detailed and the reader is referred to the above references.

An important task in the design of a safety-critical system consists in evaluating the safety integrity of an application. A method to evaluate the safety integrity of automotive applications in terms of functional units against random external disturbances has been developed [PIME06a]. The method is a three-step process that requires the evaluation of the probability of message delivery failure of the communication system, the probability of communication cycle failure, and finally the probability of failure of a functional unit. The method has been used to evaluate the safety integrity of a steer-by-wire functional unit implemented using FlexCAN yielding excellent results [PIME06a].

6.6.4 FlexCAN Applications and Summary

The FlexCAN architecture has been used to design and implement a steer-by-wire functional unit for a golf car at Kettering University [PIME06]. The design of a drive-by-wire functional unit for a lift truck at the University of Padova has been completed [BERT06] and its implementation is under way. Other implementations involving the power train functional unit of a HEV are being planned.

In this section, we have provided a brief overview of the FlexCAN architecture including its deterministic, flexible, and dependable features. FlexCAN improves CAN's deterministic behavior by incorporating a time-triggered structured in a manner similar to FlexRay but without resorting to TDMA bus access schemes and without clock synchronization. Unlike TTCAN, FlexCAN retains most of the flexibility and dependability of the CAN protocol. A set of optional features such as fail-silence enforcement, channel replication, node replication, and bus guardians increasingly improves FlexCAN's dependability thus making it suitable for safety-critical applications if so desired. The FlexCAN architecture and its protocols have been extensively simulated, and implemented in several applications.

6.7 Other Approaches to Dependability in CAN

This section encompasses a few other recent efforts to add services to CAN that, in some way, also give a contribution to improve the communication dependability. Namely, we include time-triggered CAN (TTCAN), the fault-tolerant time-triggered scheme of communication proposed in Ref. [SHOR07], Timely CAN (TCAN), ServerCAN, and fault-tolerant clock synchronization services.

6.7.1 TTCAN

TTCAN is an ISO standard [ISO01] that was developed to provide additional time-triggered transmission control to the original CAN protocol, based on a high precision network-wide time base. This reduces the jitter limitation of CAN, makes transmission instants deterministic and provides a means to carry out prompt detection of omissions at the receivers, enabling fast responses to such situations.

There are two possible levels in TTCAN, level 1 using local time (cycle time), only, and level 2 using hardware supported external clock synchronization that performs a continuous drift correction among the CAN controllers. The clock synchronization is enforced by a time master that sends specific time reference messages.

TTCAN adopts a TDMA medium access protocol. Network nodes are assigned different slots to access the bus. The sequence of time slots allocated to nodes is described in basic cycles and these, in turn, are grouped in a matrix cycle (Figure 6.16). All basic cycles in the matrix have the same duration but can differ in their slot structure. The matrix is scanned sequentially in a cycle that repeats endlessly, defining a periodic message transmission schedule. Each basic cycle starts with the transmission of the reference message.

Since TTCAN builds on top of CAN, it inherits most of its properties, including the maximum bit rate of 1 Mbps, the good bandwidth efficiency with short payloads, and the distributed arbitration mechanism. This mechanism is used for several purposes, namely to support shared slots, called arbitration windows, in which event messages can be transmitted if ready at the start of the slot, and also to tolerate deviations in

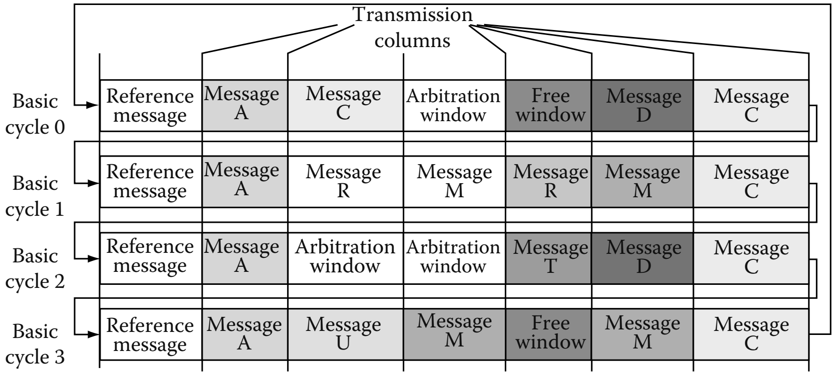


FIGURE 6.16 TTCAN system matrix with four basic cycles. (Adapted from Führer, T., Müller, B., Dieterle, W., Hartwich, F., Hugel, R., Walther, M., and GmbH, R.B., Time triggered communication on CAN. In: *Proceedings of the Seventh International CAN Conference*, Amsterdam, the Netherlands, 2000.)

the timing domain, which do not cause a direct loss of data but delayed transmissions instead. This latter aspect is, after all, shared by virtually all timed higher layer CAN protocols, increasing their robustness. Generally, nodes transmit using single-shot mode, meaning that they either succeed in starting transmission immediately or they will not transmit. This also implies that the automatic retransmission upon error is disabled.

TTCAN may operate in four modes: configuration, CAN communication, time-triggered communication, and event-synchronized time-triggered communication. Changing between modes implies returning to configuration mode. For safety reasons, this mode is also the only one in which it is possible to write onto the system matrix, thus implying a static periodic schedule at run-time.

TTCAN uses an error-containment strategy similar to that of CAN (ISO 11898-1), that is, based on error passive and bus-off states, but adds detection of scheduling errors, for example, absence of a message or collision in an exclusive slot. This information can be used at higher layers to implement other active fault confinement mechanisms. The error-passive and bus-off mechanisms are relatively slow to act, depending on the frequency and type of errors detected, but they eventually provide fail silence, according to the TTCAN specification. Similarly to CAN, bus guardians are not considered in the standard (Section 6.1).

One particular aspect that has created some controversy is the use of the single-shot transmission mode. In this case, the reaction to inconsistent scenarios discussed earlier in Section 6.2 is substantially different with respect to CAN. In fact, the absence of automatic retransmission will avoid inconsistent duplicates but will generate a much higher probability of IMOs. For this reason some authors claim that TTCAN is less suited to safety-critical applications than CAN [RODR03a].

Finally, TTCAN also provides a mechanism for time masters replication and replacement to assure a continued transmission of the reference message.

6.7.2 Fault-Tolerant Time-Triggered Communication Using CAN

Short and Pont [SHOR07] recently demonstrated an interesting alternative to improve reliable CAN communications by exploiting channel redundancy. Their scheme uses TDMA on each channel based on synchronized clocks and requires that retransmission of CAN messages due to channel errors is disabled (single-shot transmission mode). As discussed in Section 6.2.3 for the case of TTCAN, it is well known that single-shot transmission eliminates IMDs but increases the probability of IMOs. Short and Pont have demonstrated a channel redundancy management scheme where the probability of IMOs is reduced to acceptable levels.

An interesting characteristic of this solution is its strong dependency on clock synchronization. In Ref. [SHOR07] the effect that clock precision has on the bandwidth utilization is studied for their channel redundancy scheme. It is concluded that as the CAN bit rate is increased, the requirements on clock precision increase as well. For instance, it is shown that at the maximum bit rate (1 Mbps), an improvement of the precision from 100 to 10 μs yields an increment of the bandwidth utilization close to 40%.

It is also important to remark that this solution does not address some of the dependability limitations of CAN. Aspects such as error containment and support for fault tolerance are left open and should be solved with additional techniques and protocols. Nevertheless, operational flexibility as discussed in Section 6.5 seems to be incompatible with this scheme, as it relies on a static TDMA scheme in order to manage channel redundancy.

6.7.3 TCAN

The timely-CAN (TCAN) [BROS03] protocol, previously called latest-send time CAN, was developed to confine the interference caused by the automatic retransmission of CAN upon errors. It is thus a form of error containment in the temporal domain. TCAN is based on the observation that if the message transmitter knows that its message will arrive to the receivers after the deadline then it is better to drop it, saving bandwidth and reducing interference over the remaining traffic (Figure 6.17). According to this scheme, all incoming messages arrive in time or do not arrive at all.

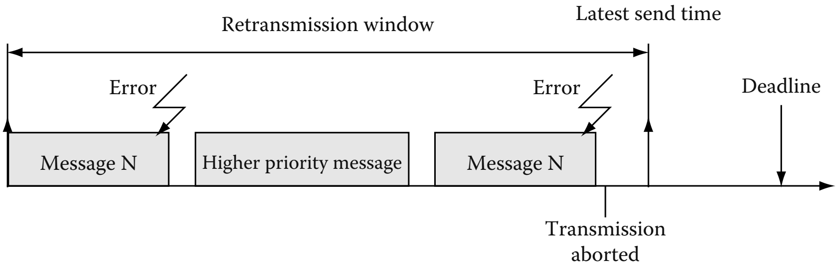


FIGURE 6.17 Typical TCAN message transmission scenario.

The bandwidth released by not transmitting messages that would arrive late can be used to facilitate timely delivery of other frames.

Each TCAN frame is associated with a transmission threshold time, termed the latest send time (LST), which is known a priori by all nodes related to that message, that is, transmitter and receivers, based on the message absolute deadline minus the frame transmission time. This requires a global time base to synchronize transmitters and receivers, implying that the absolute transmission instants are defined a priori, too. Thus, this is also a time-triggered scheme. However, the protocol copes with delays caused by possible retransmissions and interference so that the effective frame transmission may start later but always before the LST, that is, within a retransmission window starting at the specified transmission instant and ending at the LST. The precision of the global clock must be considered when defining the LST, deducing it at transmitters and adding it at receivers, to cope with clock skews. The TCAN protocol lies somewhere between CAN and TTCAN, compromising between a certain level of message retransmission capability and transmission within predictable time windows.

A response time analysis considering the bounded impact of retransmissions according to an error model has been developed. As usual in real-time analysis, when such response time is shorter than the message deadline the traffic timeliness can be guaranteed under the considered assumptions. This analysis is too costly for online use, thus requiring the message set to remain static at run-time.

Finally, several bus-guardians have been proposed for TCAN [BROS03a], which attempt to enforce a minimum intertransmission time for each message, thus improving the error-containment features of this protocol. However, some of the proposed bus guardians are inherently limited, presenting a compromise between easiness of implementation, since they use COTS components, and fault coverage, for example, in terms of confining babbling-idiot type of errors [FERR05a].

6.7.4 ServerCAN

ServerCAN [NOLT05] is another higher layer CAN protocol that was proposed to improve the robustness of communication in the presence of timing failures, for example, babbling idiots, or aperiodic sources that give no guarantees of minimum interarrival time. It is thus an improvement of error containment with respect to timing failures, reducing the interference among message streams. In this protocol each stream is handled by one server following the server scheduling paradigm in task systems, for example, periodic server, sporadic server, total bandwidth server, etc. [BUTT05]. In this paradigm, a certain transmission capacity is assigned to each message, together with adequate capacity replenishment rules. This allows limiting the bandwidth effectively used by the transmission of each message, granting prompt transmission if there is still enough capacity, or holding the transmission if the capacity is exhausted.

In order to implement the required transmission control and servers management, the protocol uses an efficient master–slave mechanism (Figure 6.18), based on that of FTT-CAN (Section 6.5), that is, organized in ECs triggered by a single master message, the TM, which encodes the identification of several polled messages. The master executes all servers to determine the current capacities and polls the messages whose

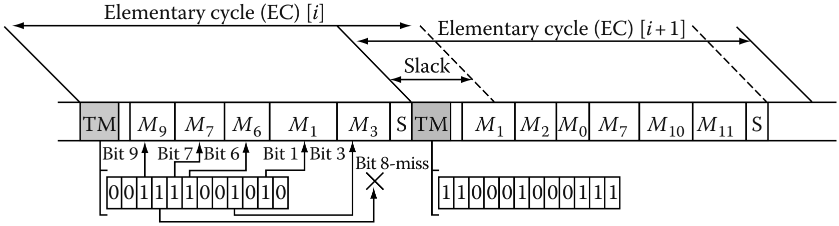


FIGURE 6.18 EC and master polling in ServerCAN. (Adapted from Nolte, T., Nolin, M., and Hansson, H., *IEEE Trans. Ind. Inf.*, 1(3), 192, 2005.)

servers still have enough capacity left. The slave nodes manage the queues of locally generated messages and transmit those that are polled by the master, only. The ECs can finish before their nominal duration if the polled messages are not ready (omissions). Notice, however, that these omissions are not errors, since they are part of the adopted traffic model, that is, aperiodic.

6.7.5 Fault-Tolerant Clock Synchronization Over CAN

It is well known that dependable distributed embedded systems may greatly benefit from a clock synchronization service, for example, to simplify mechanisms such as failure detection and redundancy management or generally to carry out synchronized actions. Some of the protocols referred in this chapter, such as CANELy, TTCAN, and TCAN, do rely on a clock synchronization service. Because of the advantages that this service brings along, its absence in the CAN standard [ISO93] has been pointed out as a CAN limitation, as referred to in Section 6.1. Therefore, clock synchronization, when required or desired in a CAN network, has to be enforced by means of an external mechanism. Moreover, whenever high-layer mechanisms are built upon the assumption of a synchronized clock, it is very important to guarantee that the clock synchronization service is reliable enough, by providing it with adequate fault-tolerance techniques. In this section, we briefly discuss some solutions that have been proposed in the literature to implement fault-tolerant clock synchronization over CAN, namely those described in Refs. [RODR98,LEE03,RODR06]. Fault-tolerant clock synchronization is also addressed in Ref. [FUHR00] but specific of TTCAN.

The solution proposed in Ref. [RODR98] constitutes one important element of the CANELy architecture, presented in Section 6.5, though it can be applied to any CAN architecture. This solution follows a classical approach in dependable distributed systems: the nodes periodically execute an agreement protocol in order to decide which one is the right clock reference and then they synchronize to this reference. This agreement protocol is performed in various rounds of message exchanges. This solution considers a wide fault model ranging from processor crash to Byzantine failures and presents the advantage of being a purely software solution. The disadvantages of this solution are that it only achieves limited clock precision (~100 μs) and that it may cause bursts of messages, which should be carefully taken into account when performing the message scheduling.

The solution presented in Ref. [LEE03] is similar to the previous one. It is also implemented in software but it reduces the number of messages required in every synchronization round. The authors claim that a precision of a few microseconds can be achieved. The main weakness of this solution is its limited fault model, which does not include inconsistent message transmissions (Section 6.2).

The specification of TTCAN level 2 [FUHR00] includes a hardware-implemented clock synchronization service, which is incorporated into the TTCAN controller. This service is based on a master–slave scheme and uses master replication. The precision achieved is in the order of one bit length. Unfortunately, the clock synchronization service of TTCAN cannot be adopted by standard CAN networks. First, because it requires a TTCAN controller (instead of a standard CAN controller) and second, because it relies on the reference message of TTCAN and thus requires the communication to be organized like the system matrix discussed in Section 6.7.1. Moreover, it only considers a limited fault model, which does not include, for instance, inconsistent messages transmissions.

The fault-tolerant clock synchronization service presented in Ref. [RODR06] was inspired by TTCAN level 2, but it is intended to overcome the drawbacks of the previously mentioned solutions. It also requires the use of a specifically designed hardware element: the so-called clock unit, but it is compatible with any CAN controller. The clock unit significantly improves the achievable precision (~4 μs) and reduces the number of necessary messages per round. This solution follows a master–slave scheme, which seems to be the preferred paradigm for clock synchronization in low-cost distributed embedded systems [IEEE1588], and also relies on master replication in order to avoid the single point of failure. Additionally, this solution introduces a number of fault-tolerance mechanisms that enforce the desired clock precision even in the presence of a wide range of channel and node faults, including inconsistent omissions and duplicates. The correctness of these mechanisms has been formally verified by means of model checking.

6.8 Conclusion

The quest for higher levels of innovation by car manufacturers, together with stringent requirements, is generating a growing interest on flexible car architectures that separate functionality from underlying control, computing, and communication architectures thus simplifying the overall vehicle design. However, this imposes a strong integration effort that has implications in all the supply chain, requiring, for example, the development and use of open standards (e.g., AUTomotive Open System ARchitecture [AUTOSAR]) and other modular approaches to software design (e.g., component-based design).

At the network level, the current trend is to use a hierarchical approach with a backbone interconnecting several subnetworks, each dedicated to one major automotive subsystem namely chassis, power train, X-by-wire, body, and infotainment. This allows achieving a high isolation between subsystems, thus reducing mutual interference, and also allows using different networking technologies, with a cost-performance ratio adapted to each case. In fact, those subsystems exhibit different requirements in terms of throughput, determinism, and dependability. Therefore, the challenge at the network level is to provide the required level of flexibility, together with the right techniques and technologies that enforce the necessary properties to satisfy the requirements with minimum cost.

CAN is a mature technology that is well known and widely used, supporting a high level of flexibility with relatively low cost, determinism, and throughput, which seems adequate for the subnetworks level referred above. However, it still presents some limitations, particularly concerning dependability, which have raised doubts on CAN adequacy to support, for example, safety-critical applications.

In this chapter, we have addressed such CAN limitations, including the so-called inconsistent communication scenarios that are typically pointed out as impairments to dependability, as well as the techniques recently developed to overcome or minimize such limitations. We also presented several techniques, protocols, and architectures based on CAN that improve the dependability of the original protocol in some aspect but still maintaining a high level of flexibility, namely (Re)CANcentrate, CANELy, FTT-CAN, and FlexCAN. While (Re)CANcentrate operates mainly at the physical and datalink layers, CANELy focuses on datalink issues and partially on higher layer, and FTT-CAN and FlexCAN are two higher layer protocols that may operate over COTS CAN controllers. Finally, we have also included a reference to a few other protocols/services that are somehow related to the topic of dependability and flexibility. Among them, TTCAN, TCAN, ServerCAN, and fault-tolerant clock synchronization focus on faults in the time domain, whereas the fault-tolerant time-triggered scheme of communication proposed in Ref. [SHOR07] improves reliable CAN communications by exploiting channel redundancy.

This collection of techniques provides networking solutions with variable levels of dependability while still maintaining the flexibility of the native CAN protocol to a large extent, thus responding adequately to the current quest for flexible car architectures. Moreover, we believe that CAN, complemented with adequate techniques such as those discussed in this chapter, is also adequate to support the most demanding subsystems, such as chassis, power train, passive safety, and even X-by-wire, and at a cost substantially lower than that of other alternatives, such as TTP/C and FlexRay.

References

[ALME02] L. Almeida, P. Pedreiras, and J.A. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Transactions on Industrial Electronics*, 49 (2), December 2002.

[ALME03] L. Almeida. A word for operational flexibility in distributed safety-critical systems, WORDS 2003. In: *IEEE Workshop on Object-Oriented, Real-Time and Dependable Systems*, Guadalajara, Mexico, January 2003.

[BARR04] M. Barranco, G. Rodríguez-Navas, J. Proenza, and L. Almeida. CANcentrate: An active star topology for CAN networks, WFCS'04. In: *IEEE Workshop on Factory Communication Systems*, Vienna, Austria, 2004.

[BARR05] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida. A CAN hub with improved error detection and isolation. In: *10th International CAN Conference*, Italy, March 2005.

[BARR05a] M. Barranco, L. Almeida, and J. Proenza. ReCANcentrate: A replicated star topology for CAN networks. In: *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005)*, Catania, Italy, 2005.

[BARR06] M. Barranco, L. Almeida, and J. Proenza. Experimental assessment of ReCANcentrate, a replicated star topology for CAN. In: *SAE 2007 World Congress*, Detroit, MI, 2006.

[BARR06a] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida. An active star topology for improving fault confinement in CAN networks. *IEEE Transactions on Industrial Informatics*, 2 (2), May 2006.

[BAUE02] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in the time-triggered architecture. In: *Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems*, Pisa, Italy, 2002.

[BERT06] M. Bertoluzzo, G. Buja, and J. Pimentel. Design of a safety-critical drive-by-wire system using FlexCAN, SAE Congress, Paper No. 2006-01-1026, April 2006, Detroit, MI.

[BOND98] A. Bondavalli, F. Di Giandomenico, F. Grandoni, D. Powell, and C. Rabejac. State restoration in a COTS-based N-modular architecture. In: *Proceedings of the First IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE Computer Society, Washington, DC, 1998.

[BOUY05] B. Bouyssounouse and J. Sifakis (Eds.). *Embedded Systems Design, the ARTIST Roadmap for Research and Development* (*Lecture Notes in Computer Science*), Vol. 3436, Springer, New York, 2005.

[BROS03] I. Broster. Flexibility in dependable communication. PhD thesis, Department of Computer Science, University of York, York, United Kingdom, August 2003.

[BROS03a] I. Broster and A. Burns. An analyzable bus-guardian for event-triggered communication. In: *Proceedings of IEEE Real-Time Systems Symposium, RTSS 2003*, Cancun, Mexico, December 2003.

[BROS04] I. Broster, A. Burns, and G. Rodriguez-Navas. Comparing real-time communication under electromagnetic interference. In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, 2004, pp. 45–52.

[BUJA05] G. Buja, J.R. Pimentel, and A. Zuccollo. Overcoming babbling-idiot failures in the FlexCAN architecture: A simple bus guardian. In: *Proceedings of 10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, 2005, pp. 461–468.

[BUJA06] G. Buja, A. Zucollo, and J.R. Pimentel. Overcoming babbling-idiot failures in CAN: A simple and effective bus guardian solution for the FlexCAN architecture. *IEEE Transactions on Industrial Informatics*, 3 (3), August 2007.

[BUTT02] G. Buttazo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51:289–302, March 2002.

[BUTT05] G. Buttazo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd edn., Springer, New York, 2005.

[CAN91] CAN Specifications Version 2.0, Robert Bosch GmbH, 1991. Available at: http://www.can.bosch.com/docu/can2spec.pdf.

[CAN04] CAN Newsletter, CAN in Automation GmbH, December 2004, Erlangen, Germany, Vol. 4, p. 84.

[CENA01] G. Cena, L. Durante, and A. Valenzano. A new CAN-like field network based on a star topology. *Computer Standards and Interfaces*, 23 (3):209–222, July 2001.

[CIA] CAN physical layer, CAN in Automation (CiA), Am Weichselgarten 26, Tech. Rep. [Online]. Available at: headquarters@can-cia.de

[ETSC01] K. Etschberger. *Controller Area Network, Basics, Protocols, Chips and Applications*. IXXAT Press, Germany, 2001.

[FERR98] P. Ferriol, F. Navio, J.J. Navio, J. Pons, J. Proenza, and J. Miro-Julia. A double CAN architecture for fault-tolerant control systems. In: *Fifth International CAN Conference*, San Jose, CA, 1998.

[FERR04] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca. An experiment to assess bit error rate in CAN. In: *Proceedings of the Third International Workshop on Real-time Networks*, Catania, Italy, 2004.

[FERR05] J. Ferreira. Fault-tolerance in flexible real-time communication systems. PhD thesis, University of Aveiro, Aveiro, Portugal, 2005.

[FERR05a] J. Ferreira, L. Almeida, and J.A. Fonseca. Bus guardians for CAN: A taxonomy and a comparative study. In: *Proceedings of the WDAS 2005, Workshop on Dependable Automation Systems*, Salvador, Brazil, October 2005.

[FERR06] J. Ferreira, L. Almeida, J.A. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza. Combining operational flexibility and dependability in FTT-CAN. *IEEE Transactions on Industrial Informatics*, 2 (2):95–102, May 2006.

[FLEX05] FlexRayŹ. FlexRay Communications System Protocol Specification Version 2.1 Revision A, 2005. Available at: http://www.flexray.com/

[FLEX05a] FlexRayŹ. FlexRay Communications System Preliminary Central Bus Guardian Specification Version 2.0.9, 2005. Available at: http://www.flexray.com/

[FRED02] L.-B. Fredriksson. CAN for critical embedded automotive networks. *IEEE Micro Special Issue on Critical Embedded Automotive Networks*, 22 (4):28–35, July–August 2002.

[FUHR00] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and R.B. GmbH. Time triggered communication on CAN. In: *Proceedings of the Seventh International CAN Conference*, Amsterdam, the Netherlands, 2000.

[GAUJ05] B. Gaujal and N. Navet. Fault confinement mechanisms on CAN: Analysis and improvements. *IEEE Transactions on Vehicular Technology*, 54 (3):1103–1113, May 2005.

[HADZ93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In: S.J. Mullender (Ed.), *Distributed Systems*, 2nd edn., ACM-Press, Addison-Wesley, Reading, MA, 1993, Chap. 5, pp. 97–145.

[HAMM03] R.C. Hammett and P.S. Babcock. Achieving 10^{-9} dependability with drive-by-wire systems. Society of Automotive Engineers (SAE) Technical Paper Series, Paper 2003-01-1290, 2003.

[HILM97] H. Hilmer, H.-D. Kochs, and E. Dittmar. A fault-tolerant communication architecture for real-time control systems. In: *Proceedings of the IEEE International Workshop on Factory Communication Systems*, Barcelona, Spain, 1997.

[HOYM92] K. Hoyme and K. Driscoll. SAFEBus, IEEE/AIAA. In: *Proceedings of the 11th Digital Avionics Systems Conference*, Seattle, Washington, DC, 1992.

[IEEE1588] IEEE-1588. Standard for a precision clock synchronization protocol for networked measurement and control systems. IEEE Instrumentation and Measurement Society, 2003.

[ISO93] ISO, ISO11898. Road vehicles—interchange of digital information—Controller Area Network (CAN) for high-speed communication, 1993.

[ISO01] ISO. Road vehicles—Controller Area Network (CAN)—part 4: Time triggered communication, 2001.

[IXXA05] Innovative products for industrial and automotive communication systems, IXXAT 2005. [Online]. Available: http://www.ixxat.de/index.php

[KOPE03] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In: *Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems, ISADS*, Pisa, Italy, 2003.

[KOPE95] H. Kopetz. Automotive electronics—present state and future prospects. In: *Digest of Papers of the IEEE 25th International Symposium on Fault-Tolerant Computing—Special Issue*, Pasadena, CA, 1995.

[KOPE97] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, MA, 1997.

[KOPE99] H. Kopetz. A comparison on CAN and TTP. Available at: http://www.tttech.com/technology/docs/protocol_comparisons/HK_1998–99 Comparison-TTP.pdf.

[LAPR01] J.-C. Laprie, A. Avizienis, and B. Randell. Fundamental concepts of dependability. Technical Report 739, School of Computing Science, University of Newcastle upon Tyne, 2001.

[LEE03] D. Lee and G. Allan. Fault-tolerant clock synchronisation with microsecond-precision for CAN networked systems. In: *Proceedings of the Ninth International CAN Conference*, Munich, Germany, 2003.

[LIMA03] G. Lima and A. Burns. A consensus protocol for CAN-based systems. In: *Proceedings of the 24th IEEE Real Time Systems Symposium*, Cancun, Mexico, 2003.

[LIVA99] M.A. Livani. SHARE: A transparent approach to fault-tolerant broadcast in CAN. In: *Proceedings of the Sixth International CAN Conference*, Torino, Italy, 1999.

[LU02] C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Special Issue of Real-Time Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23 (1–2):85–126, July/September, 2002.

[MARA06] R. Marau, L. Almeida, J.A. Fonseca, J. Ferreira, and V.F. Silva. Assessment of FTT-CAN master replication mechanisms for safety-critical applications. In: *SAE World Congress 2006*, Detroit, MI, April 2006.

[MARA06a] R. Marau, L. Almeida, and P. Pedreiras. Enhancing real-time communication over COTS Ethernet switches. In: *Proceedings of the Sixth IEEE International Workshop on Factory Communication*, Torino, Italy, June 2006.

[MIRO99] S. Miroslav and Y. Radimir. Actuator–sensor interface interconnectivity. *Control Engineering Practice*, 7 (1):95–100, January 1999.

[NOLT05] T. Nolte, M. Nolin, and H. Hansson. Real-time server-based communication for CAN. *IEEE Transactions on Industrial Informatics*, 1 (3):192–201, 2005.

[PEDR02] P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency, ECRTS'02. In: *EUROMICRO Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[PEDR03] P. Pedreiras and L. Almeida. The flexible time-triggered paradigm: An approach to QoS management in distributed real-time systems. In: *Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, Nice, France, April 22–23, 2003.

[PEDR05] P. Pedreiras, P. Gai, L. Almeida, and G. Buttazzo. FTT-Ethernet: A flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems. *IEEE Transactions on Industrial Informatics*, 1 (3), August 2005.

[PHIL] Philips. CAN bus specification 2.0. Parts A and B.

[PIME04] J.R. Pimentel. An architecture for a safety-critical steer-by-wire system. In: *Proceedings of the SAE World Congress*, Detroit, MI, 2004.

[PIME04a] J.R. Pimentel and J. Kaniarz. A CAN-based application level error-detection and fault-containment protocol. In: *Proceedings of the 11th IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, Salvador, Brazil, 2004.

[PIME04b] J.R. Pimentel and J.A. Fonseca. FlexCAN: A flexible architecture for highly dependable embedded applications, RTN 2004. In: *Third International Workshop on Real-Time Networks, Held in Conjunction with the 16th Euromicro International Conference on Real-Time Systems*, Catania, Italy, June 2004.

[PIME06] J.R. Pimentel. Testing, verification, and validation of a steer-by-wire system using DO-178B, SAE Congress, Paper No. 2006-01-1447, April 2006, Detroit, MI.

[PIME06a] J.R. Pimentel. Problem C: Calculation of Pmerr for FlexCAN. INRIA research report no. 00120905, December 2006.

[PINH03] L.M. Pinho and F. Vasques. Reliable real-time communication in CAN networks. *IEEE Transactions on Computers*, 52 (12):1594–1607, 2003.

[POWE92] D. Powell. Failure mode assumptions and assumption coverage. In: *Digest of Papers of the IEEE 22th International Symposium on Fault-Tolerant Computing*, Boston, MA, 1992.

[PRAS03] D. Prasad, A. Burns, and M. Atkins. The valid use of utility in adaptive real-time systems. *Real-Time Systems*, 25:277–296, 2003.

[PROE00] J. Proenza and J. Miró-Julià. MajorCAN: A modification to the Controller Area Network protocol to achieve atomic broadcast. *ICDCS Workshop on Group Communications and Computations*, C72–C79, 2000.

[RODR98] L. Rodrígues, M. Guimarães, and J. Rufino. Fault-tolerant clock synchronization in CAN. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[RODR03a] G. Rodríguez-Navas and J. Proenza, Analyzing atomic broadcast in TTCAN networks. In: *Proceedings of the Fifth IFAC International Conference on Fieldbus Systems and their Application (FET 2003)*, Aveiro, Portugal, 2003, pp. 153–156.

[RODR03b] G. Rodríguez-Navas, J. Proenza, and M. Barranco. Harmonizing dependability and real time in CAN networks. In: *Proceedings of the Second International Workshop on Real-Time LANs in the Internet Age*, Porto, Portugal, 2003.

[RODR06] G. Rodríguez-Navas, J. Proenza, and H. Hansson. An UPPAAL model for formal verification of master/slave clock synchronization over the Controller Area Network. In: *Proceedings of the Sixth IEEE International Workshop on Factory Communication Systems*, Torino, Italy, 2006.

[RUCK94] M. Rucks. Optical layer for CAN. In: *First International CAN Conference*, Mainz, Germany, November 1994.

[RUFI98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrígues. Fault-tolerant broadcasts in CAN. In: *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, Munich, Germany, 1998.

[RUFI99] J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' egg idea for CAN media redundancy FTCS-29. In: *The 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, June 1999.

[RUFI03] J. Rufino, P. Verissimo, and G. Arroz. Node failure detection and membership in CANELy. In: *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003, pp. 331–340.

[RUFI06] J. Rufino, P. Verissimo, G. Arroz, and C. Almeida. Control of inaccessibility in CANELy. In: *Proceedings of the Sixth IEEE International Workshop on Factory Communication Systems*, Torino, Italy, June 2006.

[RUSH03] J. Rushby. *A Comparison of Bus Architectures for Safety-Critical Embedded Systems*. SRI International, Menlo Park, CA, Contractor Report, 2003.

[SAE06] SAE Automotive Engineering International, May 2006.

[SAHA06] H. Saha. Active high-speed CAN HUB. In: *Proceedings of the 11th International CAN Conference (iCC 2006)*, Stockholm, Sweden, 2006.

[SCHM01] D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. Di Palma. Towards adaptive and reflective middleware for network-centric combat systems. *CrossTalk*, November 2001.

[SHEL04] C.P. Shelton and P. Koopman. Improving system dependability with functional alternatives. In: *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, IEEE Computer Society, 2004, pp. 295–304.

[SHOR07] M. Short and M.J. Pont. Fault-tolerant time-triggered communication using CAN. *IEEE Transactions on Industrial Informatics*, 3 (2), May 2007.

[SILV05] V. Silva, R. Marau, L. Almeida, J. Ferreira, M. Calha, P. Pedreiras, and J. Fonseca. Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In: *ETFA 2005, T8 Intelligent Robots and Systems Transaction*, Catania, Italy, 2005.

[SILV06] V. Silva and J.A. Fonseca. Using FTT-CAN to combine redundancy with increased bandwidth. In: *Proceedings of the 2006 IEEE International Workshop on Factory Communications Systems*, Torino, Italy, 2006, pp. 55–63.

[STOE03] G. Stoeger, A. Mueller, S. Kindleysides, and L. Gagea. Improving availability of time-triggered networks: The TTA StarCoupler. In: *SAE 2003 World Congress*, Detroit, MI, 2003.

[THOM98] J.P. Thomesse. A review of fieldbuses. *Annual Reviews in Control*, 22:35–45, 1998.

[THOM99] J.-P. Thomesse and M.L. Chavez. Main paradigms as a basis for current fieldbus concepts. In: *Proceedings of the FeT'99 (International Conference on Fieldbus Technology)*, Magdeburg, Germany, September 1999.

[TIND95] K. Tindell, A. Burns, and A.J. Wellings. Calculating Controller Area Network (CAN) message response time. *Control Engineering Practice*, 3 (8):1163–1169, 1995.

[TIND95a] K. Tindell and H. Hansson. Babbling idiots, the dual-priority protocol, and smart CAN controllers. In: *Proceedings of the Second International CAN Conference*, 1995, pp. 7.22–7.28.

[TOVA99] E. Tovar and F. Vasques. Cycle time properties of the PROFIBUS timed token protocol. *Computer Communications*, 22 (13):1206–1216, August 1999.

[TTPC] Time-Triggered Protocol TTP/C, High-Level Specification, Document Protocol Version 1.1. Available at: http://www.ttagroup.org/technology/specification.htm

III

Embedded Software and Development Processes

7

Product Lines in Automotive Electronics

Matthias Weber
Carmeq GmbH

Mark-Oliver Reiser
Technical University of Berlin

7.1 Introduction

The necessity to introduce and manage product lines is not at all new to the automotive industry. The complexity of the initial automotive product lines was predominantly driven by mechanical variation, that is, to maximize the reuse of common mechanical parts across the entire product range. Any reduction in the number of mechanical variants would not only reduce development and production cost, but also the cost of the after-sales phase of the automotive life cycle, by reducing the amount of spare part types, and by simplifying diagnosis and repair.

Classically, customer-visible variation was relatively limited in the industry, and mostly concerned with nontechnical customer-visible mechanical parts, for example, color and texture of material. Technical variations related to automotive electronics were offered in form of relatively few choices, for example, automatic versus manual transmission, different motor types.

For the last 10 years and for at least another 10 years to come, the automotive industry will undergo a radical change due to the advent of software in cars. Automotive electronics has become the major source of innovation. The use of software has led to an enormous increase in the number and complexity of different functions. In addition, software is driving fundamental changes in all phases of the automotive life cycle. For example, it has become commonplace that repair workshops simply exchange software on one or several electrical control units (ECUs) in order to remove some problem.

The downturn of this development is that the flexibility and ubiquity of software have led to an explosion in the number of variable artifacts. This does not only concern the customer-visible variability, but also all the little technical variations. In principle, every change in any part of the software of some sensor, actor or ECU introduces a new variant part with a different behavior at the interface. And it is not only the component that is changed, but the function itself is also changed, as well as any artifacts related to the function, for example, for testing and diagnosis.

Help in this situation must come from several sources, a major one is growing standardization (as described in several other chapters in this book); however, the central improvement must come from an overall product-line methodology that allows to optimize the number of variable technical artifacts while allowing to adapt customer-visible variability as needed by the current market situation.

Currently, no such methodology is available; however, in the areas of software product lines there are several emerging techniques that could form part of such a technology. In this chapter, we present a basic methodological framework that we think is suitable for product lines in automotive electronics, and in the context of this framework we characterize, present, and discuss selected techniques.

7.2 Characteristics of Automotive Product Lines

The methodological framework we are presenting here is itself based on common terminology and concepts of software product-line engineering. In order to make this chapter more self-contained, we begin by briefly introducing these concepts (Section 7.2.1). We then go on to discuss the characteristics of automotive electronics (Section 7.2.1).

7.2.1 Basic Concepts of Software Product Lines

Whenever a company is developing several products that share many common characteristics but also show certain substantial differences, product-line oriented development of these products can be considered [CN02,PBvdL05]. When following such an approach, the individual products are no longer developed independently from one another but instead, only a single, but *variable* product is developed (also called *product-line infrastructure*). From this variable product, the actual products (also called *product instances*) can be derived by configuration. This means that all development artifacts (e.g., requirements documents, component diagrams, Matlab (ML)/Simulink (SL) models, test-case descriptions) can be defined in a variable form, if their content varies from product to product. This is usually achieved by adding *variation points* to them and defining several *variants* for each of them.

The notion of software product lines and product-line oriented development will be introduced in detail in Section 7.3.1.

7.2.2 Characteristics and Needs of Automotive Electronics with Respect to Product-Line Engineering

Now, we will discuss characteristics and needs of the automotive domain (see also Ref. [Gri03]) which are relevant with respect to product-line engineering aspects. These characteristics can be grouped into the areas of variability, product configuration, and process-relevant characteristics.

7.2.2.1 Variability

7.2.2.1.1 Sources of Variation

There are several sources for a high degree of variation in products and product development artifacts in automotive electronics:

- Different customer needs, including functionality and pricing, within the main markets (e.g., EU, United States, or Japan) and beyond.
- Differences in required functionality and application constraints between body variants (e.g., limousine, station wagon) and drive-train variants.
- Differences in regulations and legal constraints between different countries or markets. These regulations and constraints are increasingly influencing functionality.
- New customer expectations and new technical standards originating from industrial areas that are at best partially controlled by the automotive industry. The main example here is telematics and entertainment.

For these reasons, a development project of a luxury vehicle today is based on a function kit that includes several thousand technical functions. Based on the above criteria, each of these functions needs to be selected or deselected.

This variability is further increased on the level of detailed specifications, design models, implementations, and testing artifacts. This variability partially results from functional variability and partially from different realization variants, for example, due to sensor and actuator variants.

This situation immediately implies the need for a tool that supports projects with highly complex variability models. Furthermore, concepts need to be supported to group and transparently manage the large number of decisions during product configuration [KKL$^+$98].

7.2.2.1.2 Complex Dependencies between Artifacts

In the automotive domain, it is very important to support dependencies between variants such as "needs" and "excludes" relations. There are various sources for these dependencies: they can originate from management decisions, they can be based on logical facts (e.g., a delivery vehicle without a rear window has no need for a rear wiper), or they may result from dependencies on the design and implementation level (e.g., different display variants have far-reaching consequences on the functionality offered and the human machine interface of operating the functionality). Some dependencies need to be defined explicitly, for example, in a feature model, while others can be derived from more detailed specifications or design models, for example, by appropriately defined rules. The difficulty here is to decide which dependencies need to be derived in order to make the correct decisions during product configuration.

It is necessary to manage all dependencies (defined ones and derived ones) in a central place without having to manually redefine derived dependencies. Rather, derived dependencies should optionally be made visible together with a references to their explicit or implicit definition. Two kinds of analyses need to be supported on such dependencies: analysis of logical consistency, which, if absent would preclude valid product configurations, and analysis of the objects which, based on the dependencies, needs to be added to a variant in order to reach a valid product configuration.

7.2.2.1.3 Cooperation of Heterogeneous Variability Mechanisms

We have indicated sources of variability during product development. In the automotive area, variation needs to be controlled during the entire life cycle: in the development tool chains, in the production databases, in the marketing systems, and in after-sales systems such as diagnosis and software update systems.

There exist now well-established tools for specific process aspects, for example, requirements management or function modeling. Such tools sometimes offer pragmatically grown mechanisms to support variability. This means that a comprehensive approach needs to support projects in which different variability mechanisms are used in different process steps. In order to avoid a combinatorial explosion when interfacing these mechanisms, an interface has to be defined between the variability information within a specific process step and the central variability model.

7.2.2.1.4 Different Views on Variability

A product-line oriented development methodology needs to support variability concepts relevant not only for engineering, but also for management, marketing, production, acquisition and sales, and the customer. However, the different actors and activities involved lead to different needs and expectations. Development engineers, for example, need to work with a much more fine-grained feature model than, say, management. Furthermore, changes in variability on the level of realization decisions should normally not be made visible to marketing and salespeople. Another example is the packaging of variability in order to reduce the amount of customer decisions during the configuration, or to emphasize to the customer certain combinations of functionality. It is quite clear therefore, that product-line tools need to offer and manage a broad range of views on the properties and the variability of a product line.

7.2.2.2 Product Configuration

7.2.2.2.1 Complex Configurations

Product configuration does not only occur for the final product, but also for many intermediate prototypes that are needed for concept evaluations and for system integration and testing. Frequently, the first prototype is a vehicle with a base configuration, during development, however, this base configuration is typically subject to change. For such decisions, it is crucial that tool support for product lines is able to check for all dependencies between variation points. In the automotive area, these dependencies have become too complex for manual checking.

7.2.2.2.2 Complex Resolution of Variability over Time

In the domain of automotive software development, variabilities are bound at different times of the product life cycle, for example, basic features are selected during development time, country codes are bound during production, and additional preinstalled software-features may be activated by the after-sales process. Tools need to support these different binding times for variation points.

7.2.2.3 Process-Related Characteristics and Needs

7.2.2.3.1 No Clean Separation of Domain and Product Engineering Processes

In contrary to common practice in traditional software engineering domains, development projects in the automotive domain need to keep their deadlines at almost all cost, because otherwise cost will rise dramatically due to development, production, sales, and marketing infrastructure of automotive original equipment manufacturers (OEMs). As a consequence, development projects are sometimes forced to reduce or even withdraw planned functionalities very late or to implement specific stop-gap solutions for a single model line, in order to cope with technical difficulties or to guarantee a high level of quality.

This situation is further complicated by the fact that these model-specific solutions are—in contrary to initial plans—sometimes used in future generations of this model or even in other models and can thus become new standard solutions. In most cases, it is not possible to predict in advance if this will occur. Instead, the specific stop-gap solution is for some time in competition with the originally intended solution.

Even though rigid reuse across all development projects is desirable, it is therefore impossible in practice to organize them as instances of a single product line in a strict sense. Instead, there is the need to be able to formulate a common strategy for the entire product line, which allows individual departments responsible for a part of the product range—such as the electronic platform of the Mercedes Benz C-Class—to deviate from it to some extent. Product-line methods and tools for the automotive domain must provide support for such local deviations while at the same time fostering the main goal of product-line oriented development, namely the maximization of rigid reuse of technical artifacts.

7.2.2.3.2 Synchronization with Supplier Strategies

An automotive OEM is faced with the necessity to integrate the confidential product-line strategies of a multitude of suppliers into his or her own product-line strategy. Usually the suppliers' product lines are evolved completely orthogonally from the manufacturer's product line, because a supplier is in contact with several manufacturers with often diverse strategies. When this challenge is ignored, product quality and cost are negatively influenced to a considerable degree. It is therefore essential that product-line engineering methods and tools provide means to integrate a multitude of independently developed subordinate product lines—that is, those of the suppliers—into a single higher-level product line—that of the manufacturer—without the need of disclosing all confidential details.

7.2.2.3.3 Synchronization with Change Management Processes

Present change management processes must be adjusted to a product-line oriented development. Automotive product-line methods and tools must provide support for complex change management. In particular, it must be possible to reveal the impact of changes to the product-line strategy on cost and schedule and to forward these changes into the affected development artifacts and to modify them accordingly.

7.2.2.3.4 Very Difficult Incremental Introduction

A step-by-step introduction of product-line methods, processes, and tools is necessary in many domains. But this is particularly true for the automotive domain and the prerequisites for a stepwise introduction are especially intricate here. Already the longevity of automotive systems gives cause to this problem: the life cycle of a certain generation of an upper-class model usually comprises 3 years of development, 6 years of production and sales, and 15 years of operation, maintenance, and customer service. Even subsystems of a certain model cannot be integrated in a single step in a product-line infrastructure because compatibility with legacy systems must be observed. Instead, a bottom-up approach is more realistic: at first, small and localized product lines are set up for selected subsystems (e.g., wiper or climate control) or even for individual development artifacts (e.g., requirement modules or a set of test cases); in a next step, these will be integrated in a larger, higher-level product line for a subdomain like telematics, body electronics, or motor control. Then, these will be further integrated into a comprehensive product line for a complete vehicle model. Methods and tools must allow for such a stepwise introduction of product-line techniques in a bottom-up manner.

7.3 Basic Terminology

Before discussing the overall structure of an automotive product line in the coming section and describing how variability is defined within various types of development artifacts later in Section 7.5, we first introduce basic terms and concepts of software product lines in this section. We begin this introduction with a discussion of the terms software product line and variability (Sections 7.3.1 and 7.3.2), followed by an overview of feature modeling as a special form of variability modeling and conclude it with an overview of key characteristics of software product-line methods and processes.

7.3.1 Software Product Lines

Even though there does not exist a single, generally accepted definition of the term software product line, research and practice acquired a fairly consolidated, common understanding of its meaning over the past decade. Hereby, an interesting shift in focus of software product-line research can be observed compared to an earlier understanding of the term. To illustrate this briefly, we introduce the two most influential definitions here.

In 1976, David L. Parnas [Par 76] first coined the term software product line and defined it as follows:

> A set of programs constitutes a product line whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual members.

Notably, only such software products form a software product line, that have a certain degree of commonality while at the same time showing substantial differences. This combination of commonality and variability is one of the key characteristics of a software product line. And the question of what degree of commonality is necessary in order to be able to apply product-line oriented development methods is one of the most important—and difficult—questions of the field. In case the products differ too much from one another, the overhead of describing them as members of a single product line is too high in order to gain substantial benefit from this approach.

Parnas proposed to manage a product line by composing its individual products from reusable modules or components with clearly defined interfaces, thus setting the direction of research for the coming decades. From today's point of view, this early approach to product-line development could be called a "classical" or "conventional" reuse approach. The above-mentioned shift in focus occurred in the early 1990s, which is reflected in the following definition of Clements and Northrop [CN02]:

> A software product line is a set of software products [...] that are developed from a common set of core assets in a prescribed way.

Now, the individual products are no longer each developed in parallel by composing reusable parts. Instead, the products are derived from a single, common product definition—the *product-line infrastructure*—in a *prescribed* way. Instead of describing each product separately, there is only a single description of the product line together with a definition of how individual products differ from this prototype. To achieve this, all development artifacts that are part of the product-line infrastructure (e.g., requirements documents, component diagrams, ML/SL models, test-case descriptions) are defined in variable form if their content varies from product to product.

The individual products derived from the infrastructure are then referred to as *product instances* or simply *products* of the product line.

The difference between a product-line approach and conventional reuse is illustrated in Figure 7.1. In the case of conventional reuse there is still a complete system description for each product, even if these descriptions are made up of reusable parts (cf. Figure 7.1b). How the reused assets are combined is defined for each product separately. This is not the case with product-line oriented development: there, only one, variable system description is defined (Figure 7.1c).

In summary, we can thus define a software product line as follows:

> A software product line is a set of software products that share a certain degree of commonality while also showing substantial differences and that are derived from a single, variable product definition—the product-line infrastructure—in a well-defined, prescribed way.

Sometimes the term *software family* is used as a synonym for software product line. We prefer to rather think of a software family as a very complex software product line, possibly made up of several smaller software product lines. For example, the Mercedes Benz C-Class C320 would be a typical product line while all Mercedes Benz vehicles, passenger cars, as well as commercial vehicles, together form a product family. Sometimes also the term *product population* is proposed to refer to such complex product lines [vO00].

Up to this point, we only considered software product lines to adhere closely to conventional product-line terminology as applied in the software engineering community. In the automotive domain, however, we usually do not encounter pure

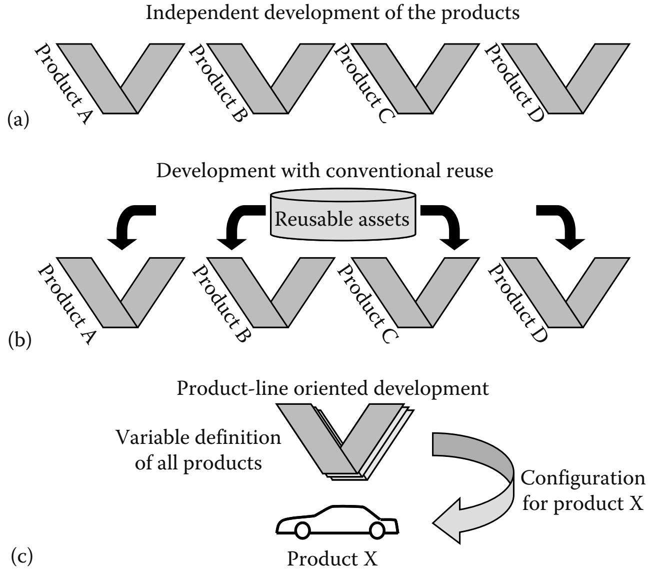

Independent development of the products
Product A
Product B
Product C
Product D
(a)
Development with conventional reuse
Reusable assets
Product A
Product B
Product C
Product D
(b)
Product-line oriented development
Variable definition of all products
Configuration for product X
Product X
(c)

FIGURE 7.1 Product-line oriented development in comparison.

software products but instead deal with software/hardware systems in which software is embedded within a hardware context, which is also subject to the overall development activity. To reflect this, we simply replace the term "software" in the above definitions by "software-intensive system" or simply "system" and thus speak of product lines/families of software-intensive systems or *system product lines/system families* for short.

7.3.2 Variability

Variability basically refers to the differences that occur when comparing the product instances of the product line to one another. In other words, whenever the product instances differ from one another in a certain aspect of their hardware or software (or both), we view this as a *variability* within the product line, whereas those hardware or software aspects that are identical for all product instances are viewed as commonality. To illustrate this, let us consider the following example:*

Example

A product line of mobile phones consists of three models, `S` (for simple), `M` (for medium), and `A` (for advanced). `S` has a black-and-white display, while the others have color displays. Only `A` has support for T9 (an advanced text input mode). All models have the same 1 mega pixel camera supplied by supplier `X`. All three models are triband devices, that is, they can establish a communication over global system for mobile communications (GSM) bands 900, 1800, and 1900 MHz.

The camera and its resolution are clearly an example of commonality. The two forms of displays and the T9 support only provided by model `A` evidently constitute variability within our product line. In case of the display, the variability affects the hardware and probably also the software, while in the other case only software is affected, because T9 is a pure software feature (we do not consider the labeling of the phone's buttons here).

Another important notion related to variability is *binding time*. Variability may be resolved—or *bound*—at various points in time during development, production, and post production. For example, the display variability will be bound before production, because it must be clear whether a black-and-white or color display must be built in. The variability of T9 support may be bound just after production, by parameterization of the phones software. This way, the same software can be used for phones with and without T9 support; the differentiation is achieved by post production configuration. The binding time can be different for each variation within the product line.

As a special case, variability may also be bound at runtime. Such variability is then called *runtime variability*. The triband functionality is an example for this: all three models are shipped with this functionality, which means they are all equipped with

* Since we will concentrate on the automotive domain in the more detailed examples below, we chose an example from a different domain here.

hardware capable of using the three GSM bands mentioned above and with software that can select the appropriate band or switch from one to another if necessary. Therefore, according to the above definition of variability, we ought to view this as commonality, because the hardware and software realizing triband functionality are identical in all three models. However, there actually is some form of variability, because the GSM band used for communication changes. Runtime variability does not result in a difference in the product instance's hardware or software, but instead constitutes a functionality of the products, a common functionality in fact. Product-line engineering approaches differ in whether they view runtime variability as a variability of the product line or not. The rationale for treating runtime variability just like ordinary variability is that the only conceptual difference is the binding time. During the early phases of commonality/variability analysis it may not be clear whether some variability will be bound prior to runtime or at runtime. Also, what may be runtime variability for some models can be ordinary variability for other models. To put it simply, the binding time determines whether some variability is realized through a variation of the product instances' hardware or software (binding prior to runtime) or if it has to be realized through a special, common functionality of the product instances (runtime binding).

7.3.3 Feature Modeling as a Form of Variability Modeling

In general, *variability modeling* is aimed at presenting an overview of a product line's commonality and variability. Depending on the form of variability modeling, commonality may either be addressed only indirectly, that is, everything that is not explicitly defined to be variable is implicitly defined to be common, or directly, that is, certain aspects are explicitly defined to be common in order to highlight them and to document the decision to make them common. In order to stress the fact that variability modeling at least indirectly addresses also commonality, sometimes the term *commonality/variability modeling* is preferred.

As is the case for all modeling activity, variability modeling tries to achieve its goals by way of abstraction. This means that some aspects of the entire information related to a product line's variability are deliberately left out of consideration in order to reduce the amount of information to a level that is manageable and that puts emphasis on the important aspects while hiding unnecessary detail.

The information captured in a variability model then serves as a basis for defining variability within the artifacts that make up the product-line infrastructure as well as for configuring individual product instances and deriving them from the infrastructure. For example, consider a ML/SL model in which a certain block $\mathtt{B}_1$ has to be replaced by some other block $\mathtt{B}_2$ depending on the selected product instance. It is the responsibility of the product-line's variability model to provide a basis for configuring individual product instances (i.e., to select and define them) and defining when $\mathtt{B}_1$ is to be used without replacement and under what circumstances $\mathtt{B}_1$ needs to be replaced by $\mathtt{B}_2$. How this is done in detail highly depends on the variability modeling technique being applied. We will describe this in detail for feature models in Section 7.5.

Roughly, three forms of variability modeling can be distinguished: *feature modeling*, *decision tables*, and *decision trees/graphs*. Figure 7.2 shows an excerpt of a decision

ID	Description	Subject	Resolution	Effect
	—			
3	Does the phone have T9 support ?	T9	Yes	Step 6 of use case send message is obligatory; extension 6a of use case 'send message' is obligatory
			No	Step 6 of use case 'send message' is removed
	—			

FIGURE 7.2 Excerpt of a sample decision table. (From Muthig, D., John, I., Anastasopoulos, M., Fonster, T., Dörr, J., and Schmid, K., Gophone—a software product line in the mobile phone domain. IESE_Report 025.04/E, Fraunhofer IESE, March 2004.).

table as presented in Ref. [MJA$^+$04], a case study of the PuLSE approach devised at Fraunhofer IESE, Kaiserslautern, Germany. A decision table usually refers to one or more variable development artifacts, in the example this is a use case diagram for the use case "send message" (not shown). Each line in a decision table represents a decision to be taken in order to configure the corresponding variable artifact(s) of the table. Each such decision has

- An ID (a plain number in the example)
- A question that formulates the decision to be taken
- A subject used to group several semantically related decisions
- A list of possible resolutions, that is, possible answers to the question
- One effect per resolution that describes how the corresponding variable artifacts have to be changed in order to configure them in alignment to the decision taken

The number and precise meaning of each column in a decision table varies from one approach to another, but the example shown here illustrates the basic idea of decision tables. Constraints are an example for such other important properties of decisions. With constraints it is possible to define interdependencies between decisions to restrict the available resolutions depending on decisions taken earlier or to hide decisions when they are no longer valid because of some other decision taken earlier. For example, if the decision "Does the phone have a camera?" was answered with "no," the decision "What is the camera's resolution?" is no longer valid and can be hidden during configuration. Likewise, decisions in decision tables can sometimes be organized hierarchically.

Similarly, decision trees define decisions to be taken in order to configure one or more variable artifacts. However, the decisions are represented and arranged graphically. Figure 7.3 shows a small example of such a decision tree. The advantage is that some selected dependencies between the decisions can easily be defined in that way. For example, the fact that decision "What is the camera's resolution?" is invalid in case the camera is dismissed altogether is clearly visible in the tree. Also the number of possible product configurations is easily detectable because each leaf in the decision tree corresponds to exactly one product configuration. However, this also points at an important problem of decision trees. They tend to become extremely large in

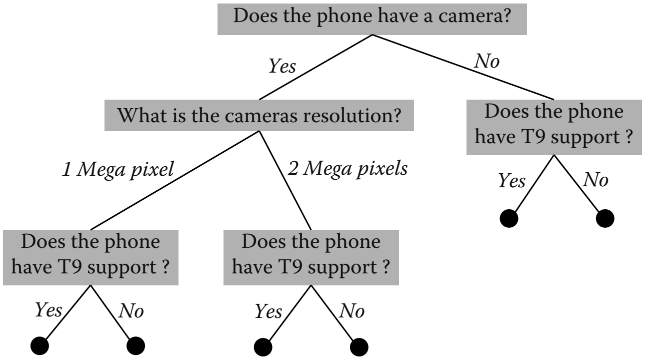

Does the phone have a camera?
Yes
No
What is the cameras resolution?
Does the phone have T9 support ?
1 Mega pixel
2 Mega pixels
Does the phone have T9 support ?
Does the phone have T9 support ?
Yes
No
Yes
No
Yes
No

FIGURE 7.3 Example of a decision tree.

complex cases. This can be avoided by using directed acyclic graphs instead of trees (i.e., a "tree" in which a node may have more than one parent).

In the remainder of the overview of automotive product lines given in this chapter we do not consider decision tables and decision trees in more detail. Instead, we will concentrate on feature modeling, arguably being the most popular form of variability modeling.

7.3.3.1 What Is a Feature?

Before presenting feature modeling in detail, we first give a short discussion of the term *feature*, being the basis of all feature modeling techniques. When speaking to engineers and practitioners during the early phases of adopting feature modeling and product-line techniques, we often recognize a considerable uncertainty and discontent with the notion of features and feature modeling. The main criticism is that the term feature has a very fuzzy, unclear meaning or that there are many different forms of understanding the term feature within the automotive industry or even within a single company alone. This often leads to attempts to provide the term feature with a more concrete, specific meaning, for example, "a component," "a function," or "a customer-visible functional requirement." While such specializations or clarifications can be of value in certain circumstances, we believe that they are very problematic on a general, company-wide level (or even beyond) and lead to a misuse of feature modeling, because the broad meaning of the term feature is closely related to the strength of the approach.

Just as variability modeling in general, feature modeling is also aimed at presenting an overview of the commonality and variability between the products of a product line and at supporting product configuration. But in contrast to decision tables and decision trees/graphs, feature modeling does not focus on the decisions to be taken during configuration and the resulting effects on the variable artifacts. Instead, the focus of feature modeling directly lies on the differences and similarities of the product-line's individual products. More concretely, feature models list all important characteristics of the individual products and state whether these characteristics are common to all products—that is, each and every product shows the given characteristic—or are variable from one product to another—that is, some products show this characteristic while others do not. Since the characteristics in which the products can differ from one another can be of very diverse nature, we need a substantially abstract term to refer to these characteristics, namely the term feature. And this is the reason why it is a good thing that it has such a broad meaning.

Therefore, the term feature can be defined as follows: A feature is a characteristic or trait in the broadest sense that an individual product instance of a product line may or may not have.

Several conclusions can be stated:

- A feature is either present in a product instance or is not; this means during product configuration it can be selected or deselected.
- A feature does not necessarily correspond to a subsystem or component (features may correspond to a subsystem, for example, "climate control" or "antiblocking system," but this need not be the case, for example, for a feature such as "low energy consumption").
- A feature need not be customer visible.
- A feature is not necessarily a functional requirement.

In summary, the extreme broadness of the term feature is due to the high level of abstraction at which feature modeling takes place. This is the strength of feature modeling and otherwise—if the term feature would have a more precise meaning—feature modeling would not be able to serve its purpose.

7.3.3.2 Basic Feature Models

The purpose of feature models is to show the common and variable features of the product-line's product instances in a graphical presentation. Basic feature models in the form of feature trees were introduced by Kang et al. [KCH+90]. Figure 7.4 shows an example of the same small feature tree in two common notations.

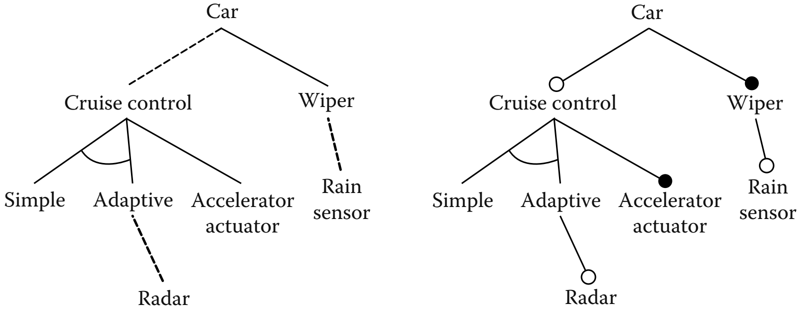


FIGURE 7.4 Example of a basic feature model in two alternative notations.

The features are represented as the tree's nodes, for example, CruiseControl or Wiper. Sometimes, the root node is seen as a special entity and referred to as the *concept*. But recently this became unpopular because it introduces a special case without true need.

Features are hierarchically structured: a *parent feature* may have one or more *child features*, which are connected with lines from parent to child. On a semantic level, this means that a child feature may only be present if the parent feature is. Therefore, the parent–child relations and the feature tree's hierarchy induced by them can be seen as a special notation for some—not all—dependencies between features.

Child features that need to be present in all product instances that contain their parent are called *mandatory features* and are denoted with a solid line to their parent or a filled circle (e.g., Wiper). It is very important to notice that mandatory features are not in all cases obligatory in the sense that they are present in all product instances of the product line. For example, Wiper is present in all product instances while AcceleratorActuator is only present in such products that have a CruiseControl (but in all of these). As a general rule, a mandatory feature represents a commonality, that is, it is present in all product instances, if and only if all its ancestors are mandatory. Conversely, *optional features* are features that are present only in some product instances that contain their parent and thus need to be directly selected or deselected during configuration. They are denoted with a dashed line to their parent or an empty circle (e.g., CruiseControl, Radar, RainSensor). Optional features never represent commonality. In addition, two or more children of the same parent that mutually exclude each other are called *alternative features*. Their parent–child relations are connected with an arc (e.g., Simple and Adaptive as two alternative forms of the Wiper). Precisely speaking, of two or more alternative features exactly one must be present in a product instance if and only if their parent is present. Therefore, during configuration, when the parent is selected, one of the alternative children must be chosen.

Consequently, the two semantically identical feature models shown in Figure 7.4 are to be interpreted in the following way: A car always has a wiper and may optionally have a cruise control. The wiper may come with a rain sensor. The cruise control always needs an electrical actuator in order to be able to influence acceleration. There are two alternative forms of cruise control: a simple one that simply keeps acceleration on a constant level and an adaptive one that keeps vehicle speed at a constant level (e.g., by increasing acceleration when the car drives up a slope). In addition, the car may be equipped with a radar. In that case, vehicle speed will be reduced when the distance to the car in advance falls below a certain threshold. Note that not all information given here is captured in the feature model; for example, the meaning of the advanced cruise control is not obvious. Such information will be provided in a feature's textual description (not shown in the figure).

Finally, additional dependencies between features may be defined in a feature tree in the form of *feature links*. These usually serve to further constrain the set of valid configurations. For example, if the radar and the rain sensor used the same installation space in the vehicle's body, this could be defined with a feature link denoted by a bidirectional arrow between RainSensor and Radar labeled with "excludes." Feature modeling techniques differ a lot in what types of dependency links they provide and some simply leave this open. Typical types of dependencies are "excludes" (bidirectional) and "needs" (unidirectional). If feature A excludes B then A may not be present if B is present and vice versa. If feature A needs feature B then A may not be present if B is missing but B may be present without A. Other feature links may give an advice to the user during configuration, for example, if feature A "suggests" B this means you can select A without B, but you should have a good reason to do so.

Despite the common tree form, feature trees are very different from decision trees. First, the nodes represent characteristics in which the product instances differ from one another instead of questions to be answered during configuration. While features may be interpreted as questions (e.g., the feature `CruiseControl` may be interpreted as the question "Does the car have a cruise control?"), this is only true for optional features. Second, a configuration of a feature tree is given by stating for each feature whether it is selected or not, whereas a configuration of a decision tree is given by selecting exactly one leaf node (or a full path from the root to a leaf). Third, decision trees define an order in which decisions are to be taken and thus prioritize the decisions, which has a significant impact on the actual effects of constraints.

In the remainder of this section, several advanced feature modeling concepts will be described. These can be seen as an optional add-on to basic feature modeling.

7.3.3.3 Cardinality-Based Feature Modeling

An important extension to basic feature modeling is *cardinality-based feature modeling*. According to this approach, each feature is assigned a cardinality similar to the cardinalities in unified modeling language (UML) class diagrams (e.g., `[0..1]`, `[1]`, `[0..*]`, `[0..4,8..12]`). A cardinality of `[0..1]` means the corresponding feature is optional (e.g., `CruiseControl` and `Radar` in Figure 7.5), `[1]` means the feature is mandatory and a maximum cardinality above 1 means the feature may appear more than once in a single product instance. Several child features of the same parent may be grouped in a *feature group*. These feature groups also have a cardinality, stating how many features of the corresponding group must/can be selected for a single product instance. For example, a group cardinality of `[1]` means that exactly one of the group's features needs to be selected whenever their parent is selected (e.g., `Simple` and `Adaptive` in Figure 7.5). This corresponds to alternative features in basic feature models.

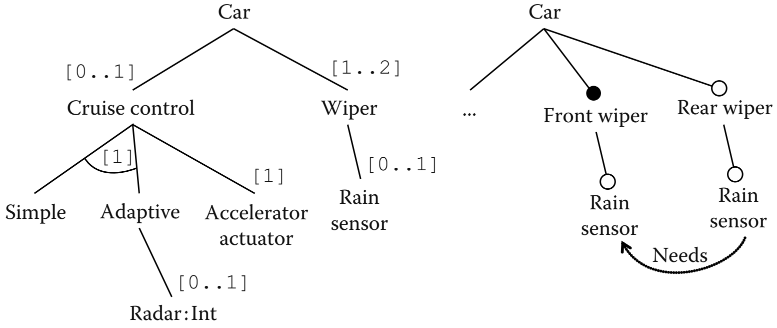


FIGURE 7.5 Example of a feature model with advanced concepts.

Apart from basing the concepts mandatory, optional, and alternative on the single, common concept of cardinalities, the main contribution of cardinality-based feature modeling is to be seen in features with a maximum cardinality greater than 1, which are called *cloned features*. During configuration, such features may be selected more than once. In that case, the cloned feature's descendants may be configured separately for each selection of the cloned feature. For example, feature `Wiper` in Figure 7.5 must be selected at least once (the front wiper) but may also be selected twice (a front and a rear wiper). In the latter case, the decision whether to equip the wiper with a rain sensor can be taken separately for the front and rear wiper. In the box on the right side of Figure 7.5, this is illustrated with the means of basic feature modeling. However, notice the subtle differences between the two presentations: the cardinality-based notation does not include the information that the two wiper features represent the front and the rear wiper and that the mandatory one is the front wiper. In addition, dependencies between the two wiper configurations—for example, the rear wiper may only be equipped with a rain sensor if the front wiper also has one, as defined by the "needs" link in the figure—cannot be expressed easily with cardinality-based feature modeling.

For maximum cardinalities of two or three and sometimes maybe four, the concept of cloned features can easily be exchanged with basic feature modeling concepts. For greater top cardinalities this is also possible in principle, but the feature models grow very large and highly redundant in these cases. For a top cardinality of * (infinite, i.e., the feature may be selected any number of times) this is no longer possible, of course. Therefore, cloned features are primarily intended for other things than a front and a rear wiper. A typical example where cloned features become particularly useful is the configuration of several ECUs in a controller area network (CAN) together with their operating systems, installed drivers, and software tasks on application layer.

7.3.3.4 Features with Several Parents

An advanced feature modeling concept with similar objectives and consequences as cloned features are *features with more than one parent*. With this concept, feature trees are turned into directed acyclic graphs. The child feature can be optional with respect to one parent while at the same time being mandatory with respect to another parent. A feature f with n parents that is mandatory with respect to n_{man} of these parents ($n_{\text{man}} \leq n$) can be interpreted as a feature with a cardinality of $[n_{\text{man}} \ldots n]$. For example, `Wiper` in Figure 7.6 is a child of both `Windscreen` and `RearWindow`. The windscreen always has a wiper, whereas the wiper for the rear window is optional. Just as is the case with cloned features, the feature `Wiper`, when selected more than once, will be configured separately each time it is selected. This means that in the example the wiper of the windscreen may have a rain sensor while the one at the rear window may not have one and vice versa.

Most problems of cloned features described above also apply to features with multiple parents. An important difference is that multiparent features are provided with more semantic meaning, because they are anchored below different parents, as can be seen in Figure 7.6.

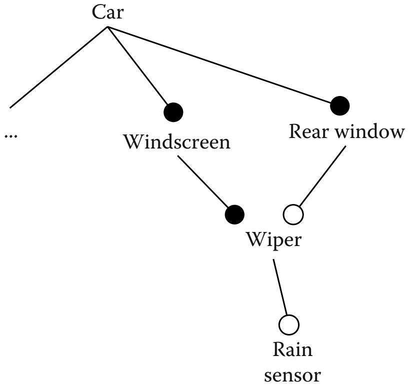


FIGURE 7.6 Example of a feature with more than one parent.

7.3.3.5 Parameterized Features

Furthermore, a feature may have a single attribute of a certain type and is then called an *attributed feature*. This attribute is of an entirely different nature than a feature's basic attributes like name and description: while the values of the latter are specified during the definition of the feature tree, the value of an attributed feature is specified during the process of feature selection and is thus becoming a constituent of the product configuration, not the feature model definition. Consequently, an attributed feature not only has the states "selected" or "not selected" during product configuration—meaning that it will be built into the product or not—but also a value that is set if and only if the feature is selected and may be a string, integer, or float. Other types are also conceivable.

Unfortunately, this common terminology makes it easy to confuse the different kinds of attributes, namely basic attributes (such as name and description) and nonbasic feature attributes. Therefore, we favor referring to the nonbasic attributes as *parameters* and to features having such parameters as *parameterized features*.

An example of a parameterized feature is given in Figure 7.5. Feature `Radar` is supplied with a type declaration of `Int` (for integer). When during configuration the radar feature is selected, an integer value has to be provided to specify the minimum distance allowed. Again, the fact that the parameter's value has this meaning is captured in the feature's documentation only.

Usually, feature parameterization is defined such that a single feature may only have a single parameter (not several), which frequently is a source of criticism of automotive engineers. The rationale for this restriction is that when a feature needs more than one parameter, it is preferable to add a child feature for each parameter instead of directly attaching the parameters to a single feature. Otherwise, information captured in the separation of the individual parameters would be "hidden" in the list of parameters even though this is usually information suitable to be captured in the feature model itself. According to our experience, this is true in most cases; but there are situations where the additional child features become very artificial. Therefore, from our point of view, there are good reasons for both solutions and thus it is mainly a matter of taste.

7.3.3.6 Configuration Decisions

The classical application domains for product-line engineering are standard desktop application programming, operating system development, and similar domains. In these domains, feature models are usually configured interactively when a particular product instance is needed, either directly by the customer or by an engineer in close cooperation with the customer. In the automotive domain, and arguably in many other embedded systems domains, this is not feasible. Feature models of automotive manufacturers can easily comprise several hundred or thousand features, which are partly purely technical and not visible to the end customer. Instead of configuring such feature models manually, it is necessary to define in advance for each feature in what context and under what circumstances it will be selected.

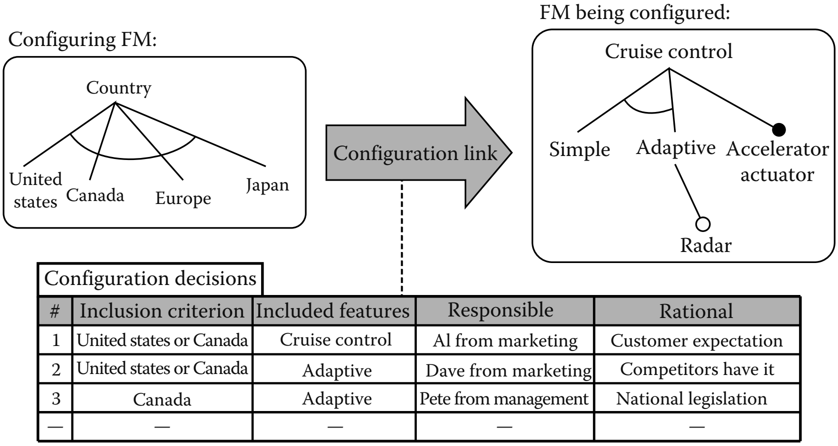
In simple cases, this can be done with a logical expression attached to each feature. These expressions are called *selection criteria*. A feature will then be selected if and only if its selection criterion evaluates to true. While being completely sufficient for simple cases, the selection criteria become extremely difficult to maintain for complex feature models. The reason for this is that individual considerations that drive the configuration definition are often spread across many different features' selection criteria and that several such considerations that affect a single feature's selection criterion are compiled into a single logical expression and can then no longer be distinguished. For complex industrial product lines this approach is therefore infeasible.

A more advanced concept for feature model configuration is *configuration decisions*. They allow to define the configuration of one feature model in terms of the configuration of another feature model in a highly scalable form. Such a link between a configuring and a configured feature model is called a *configuration link*.

An example of such a configuration link is illustrated in Figure 7.7. Let us assume that there are three considerations that influence the configuration of the cruise control feature model:

1. All North American cars must have a cruise control, because this feature is expected by customers in this market as standard equipment (marketing decision).
2. All North American cars must be equipped with an adaptive cruise control because our main competitor has it as standard equipment (marketing decision).
3. Canadian cars must include adaptive cruise control. National legislation requires this (management decision).

This situation is modeled as three configuration decisions as shown in the table in Figure 7.7. Each configuration decision has an *inclusion criterion* stating for what product instances the decision is valid. Such an inclusion criterion defines a set of product instances, called a *product set*. Next, for each decision several features of the configured feature model are listed that are included or excluded (not shown in example) in all product instances that are in the decision's product set. Finally, each



#	Inclusion criterion	Included features	Responsible	Rational
1	United states or Canada	Cruise control	Al from marketing	Customer expectation
2	United states or Canada	Adaptive	Dave from marketing	Competitors have it
3	Canada	Adaptive	Pete from management	National legislation
—	—	—	—	—

FIGURE 7.7 Defining feature configurations with configuration decisions.

decision is supplied with a person in charge and a rationale. When a certain configuration of the configuring feature model is provided, then the configuration of the configured feature model can be derived by combining the configuration decisions that are valid for the product instance that corresponds to the configuration feature model's configuration.

The benefit of documenting the individual configuration decisions separately becomes obvious when considering the redundancy of the considerations in the example. When using one selection criteria per feature, the three considerations would all be compiled into a single rule: North American cars are equipped with adaptive cruise control. However, when one of the considerations changes or ceases to apply, the clear separation of concerns is of great importance in order to establish the effect of this change on the configuration in general. For example, when the competitor changes his product range and no longer offers adaptive cruise control as standard equipment, then configuration decision no. 2 becomes invalid. In this case it is very important to know that there were other reasons to ship North American cars with adaptive cruise control. Similarly, when new considerations come up, it is possible to detect conflicts with existing ones; then the rationale of the conflicting configuration decision can be consulted or the person in charge can be contacted in order to resolve the conflict.

Many additional properties of configuration decisions are possible, for example, prioritization or limitation of validity in time.

7.3.3.7 Other Advanced Feature Modeling Concepts

Many other concepts for feature modeling have been proposed in the literature. For example, the edges of the feature tree—that is, the parent–child relations—may be typed in order to specify whether the child is a specialization (`Simple` and `Advanced` in the above examples) or a part of the parent (`CruiseControl` and `Wiper`). Also, features may be organized in several layers of refinement [KKL+98]. Further details of advanced feature modeling concepts are beyond the scope of this chapter.

7.3.4 Discussion: Feature Modeling for the Automotive Domain

The various feature modeling concepts described above are of very different usefulness in the automotive domain. As a rough guidance in general, we recommend to confine oneself to basic feature modeling whenever possible. The only exception is feature parameterization, which is probably useful or even indispensable in most use cases of automotive feature modeling and does not introduce too much additional modeling complexity. Cloned features are very useful in some situations (e.g., ECU network configuration, as described above), but should be used sparingly and only at those points in a large feature model, where they are truly necessary, due to the great complexity of this concept. Configuration decision modeling is of use whenever a complex feature model cannot be configured interactively each time a variant needs to be generated but instead the configuration has to be defined in advance.

Unfortunately, support for cloned features and configuration decisions is presently very limited in commercial tools for variability management.

7.4 Global Coordination of Automotive Product-Line Variability

Since a product-line's infrastructure usually consists of a multitude of variable artifacts, there is the need for a centralized coordination of variability across all these artifacts on a global level. This is particularly true for the automotive domain with its highly complex product lines and families. There are two schemes that can be applied for such a global variability coordination. They will be described in this section.

As a basis for this discussion, consider the following situation. All information of relevance during the development process is captured in a multitude of development artifacts (e.g., requirements documents, component diagrams, ML/SL models, test-case descriptions). In Figures 7.8 and 7.9, these artifacts are depicted with document symbols on the gray V, which symbolizes the development process. Each of these artifacts can be defined in a variable form if its content varies from one product instance to another. This is exemplarily shown in Figures 7.8 and 7.9 for the enlarged artifact on the right by the empty rectangle representing a variation point (i.e., a point at which variability occurs within an artifact; a "place-holder" for variability) and three variants that may be inserted at this point. How precisely variability is defined within an artifact is highly dependant on the type of artifact and the approach being used; this will be discussed in more detail in Section 7.5. Here, we concentrate on how to organize and manage the complex variability across the numerous artifacts of the product line.

In short terms, this means that two different levels of variability management need to be distinguished:

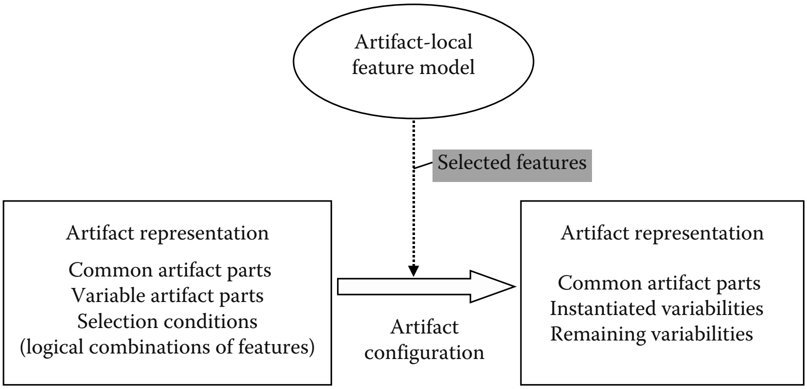


FIGURE 7.8 Product-line coordination with a core feature model.

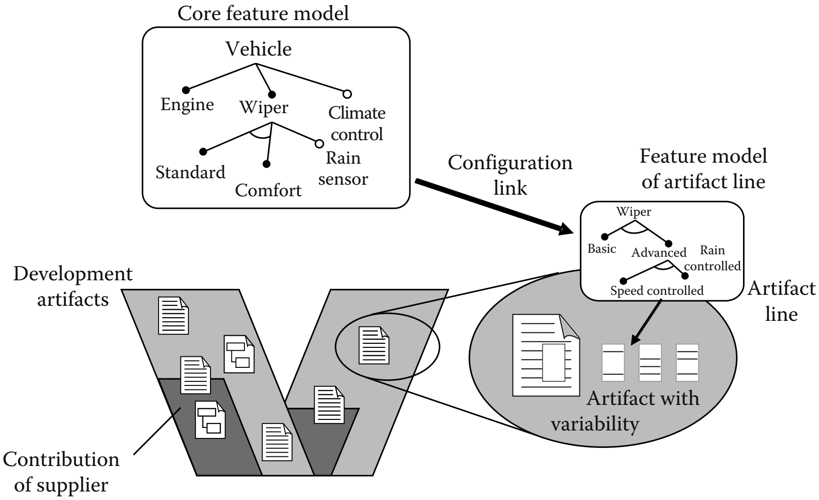


FIGURE 7.9 Product-line coordination with a core feature model and artifact lines.

1. Variability on artifact level, that is, defining that and how an artifact's content varies from one product instance to another (Section 7.5)
2. Variability on product-line level, that is, the variability of all product instances from a global perspective (remainder of this section)

7.4.1 Coordination of Small- to Medium-Sized Product Lines

One way of achieving the global perspective is to introduce a single *core variability model* for all artifacts (Figure 7.8). Feature models are especially suited for this purpose due to their high level of abstraction. This core variability model is then used as a basis to define the binding of variability in all artifacts, which is shown in

Figure 7.8 in form of the arrows from the core feature model to the variable artifacts. This means that for each variation point within an artifact, the specification of when to use which variant is formulated in terms of configurations of the core feature model. For example, consider the fictitious ML/SL model mentioned above, in which a block $\mathtt{B}_1$ needs to be replaced by a variant $\mathtt{B}_2$ for some product instances. In such a case, it would be defined that $\mathtt{B}_2$ is used if, for example, the feature `RainSensor` is selected and that otherwise $\mathtt{B}_1$ remains unchanged. This way, the entire variability within the product-line's artifacts is related to a single configuration space represented by the core feature model. And when a complete configuration of the core model is given, that is, a certain product instance is chosen, the configuration of all artifacts can be derived from that.

This organizational pattern is well suited for small- and medium-sized system product lines that are developed by only a small number of teams or departments within a single company and that comprise only a moderate number of artifacts of low or medium complexity. Product lines of automotive suppliers offering clearly delimited subsystems often fall into this category. For example, a supplier offering a product line of rain sensors or of climate controls can probably employ this scheme of product-line coordination.

7.4.2 Coordination of Highly Complex Product Lines

For several reasons, the organizational pattern described in the previous section cannot be used for highly complex product lines, such as an automotive manufacturer's product range:

1. The artifacts that make up the product line are not developed by a single company. Instead, subsystems are developed and supplied by other companies and need to be integrated into the overall product line.
2. Even within the same company, many different departments are responsible for the various artifacts, due to the overall complexity of the complete system.
3. Due to this complexity of the organizational context in which development takes place (i.e., departments within the company plus external suppliers), a multitude of different methods, tools and processes are involved in the development.
4. Both artifacts and subsystems have diverging life cycles and scope. A subsystem like a wiper or a brake-by-wire system is usually not only built into several vehicle models produced in parallel but also into several consecutive generations of these models; and often, during the production of one generation of a vehicle model, such a subsystem may even be replaced by another, that is, the introduction of a new subsystem does not necessarily coincide with the introduction of a new vehicle model. For example, the wiper control may be used in models A and B while the antiblocking system is employed in models B, C, and D. Similarly, the antiblocking system may be replaced by a new type during the lifetime of one generation of vehicle model B.

Feature modeling has great potential especially in such intricate development contexts: features may serve as a link between management, marketing, and development in a single company and as a link between companies to facilitate communication, thus becoming the core of all variability and evolution management.

However, for the above reasons it is not possible to directly relate all variable artifacts to a single, global feature model. Instead, each development artifact is organized as its own small product line. Similarly, several artifacts may be combined and may be managed together as a single, small-sized product line. Of course, the instances of these subordinate product lines are different in nature from the instances of the overall product line: in the first case we have an initialized, nonvariable development artifact, such as a test-case description or a requirements specification, while in the latter case we actually have a product, that is, an automobile. To emphasize this fact, these small-sized "product lines" of development artifacts are referred to as *artifact lines*.

The main property distinguishing an artifact line from a simple variable is that the artifact line is provided with its own local feature model (Figure 7.9). This feature model is used to publish an appropriate view of the artifact's variability to the actors interested in instances of this artifact. This makes the one or more artifacts in the artifact line independent from the global core feature model of the overall product line.

The link between the artifact line and the overall product line is achieved by defining the configuration of the artifact line's feature model as a function of the configuration of the core feature model of the overall product line, as illustrated by the solid arrow in Figure 7.9. On a conceptual level, these links can be realized with configuration links as introduced above. Then, whenever a configuration of the core feature model is given, the configuration of all artifact lines can be deduced from it. In most cases, these links are directed in the sense that configuration can be propagated from the core feature model to those of the artifact lines, and not the other way round.

Artifact lines may, in turn, be composed of other, lower-level artifact lines. To achieve this, the feature models of the lower-level lines are linked to the top-level line's feature model in exactly the same way as described before for an artifact line's feature model and that of the overall product line. In this way, variability exposed by the lower-level artifact lines may be partly hidden, diversely packaged, or presented in a different form.

Consequently, two sorts of variability information are hidden by the artifact line's feature model: first, the details of how variability is technically defined within the specific artifact (e.g., with explicitly defined variation points together with variants for them or with aspects that may optionally be woven into the artifact), and second, in the case of composite artifact lines, the details of how variability is exposed by the lower-level artifact lines. Both cases of concealment are referred to as *configuration hiding*. This configuration hiding is key to supporting the diverging life cycle of individual development artifacts as well as the complex network of manufacturer–supplier relationships described above. In addition, such a hierarchical management of variability is an effective instrument to reduce the combinatoric complexity of product-line engineering, because the artifact local feature models can reduce the complexity of variability within the artifact to a level that is appropriate for its use within the overall product line.

Instead of having a single core feature model, it is also conceivable to hierarchically decompose this core model into several feature models linked by way of the concept used to link the artifact lines' feature models to the core. For example, two core feature models could be used to distinguish a customer viewpoint and an engineering viewpoint on variability, as explained in Ref. [RW05].

Another approach for coping with highly complex product lines, called *subscoping*, follows a different idea. Instead of hierarchically decomposing the infrastructure of a complex product line, the line is replaced by several subordinate product lines, called *sublines*, each having a reduced scope with respect to the overall line, thus substantially reducing the complexity of the variability within each such subline. These sublines can then be developed relatively independently from one another but at the same time dedicated technical concepts allow for a strategic management of the overall product line on a global level. More details on this approach are beyond the scope of this introductory chapter and can be found, for example, in [ReiWeb07].

Finally, it should be noted that many details of the coordination of highly complex product lines as well as of the related approaches and techniques are still not completely solved and remain a challenging subject of further research.

7.5 Artifact-Level Variability

In the previous section, we have presented concepts for managing variability on a global coordination level. In this section, we discuss managing variability on artifact-local level and the relation between artifact-local level and global coordination level.

7.5.1 Basic Approach

When configuring an artifact, for example, a requirements specification, an analysis model, or a set of test cases for a specific product (or product set), a mechanism must be provided to select a variation point based on the decisions leading to this product (or product set).

As discussed in Section 7.3.3, there are many approaches that can be used as a basis for configuration. Here we consider a feature-based configuration of artifacts, as presented in Figure 7.10. In this approach a condition must be attached to each variable artifact part to describe the combination of feature selections under which the variable artifact part is selected. In the most simple case, there is a mapping from each variable part to a single feature. In general, however, more complex conditions may be necessary.

7.5.2 Difficulties Related to Artifact-Local Variability

Unfortunately, in the world of individual development artifacts, some difficulties have to be faced when trying to introduce mechanisms for representing variability:

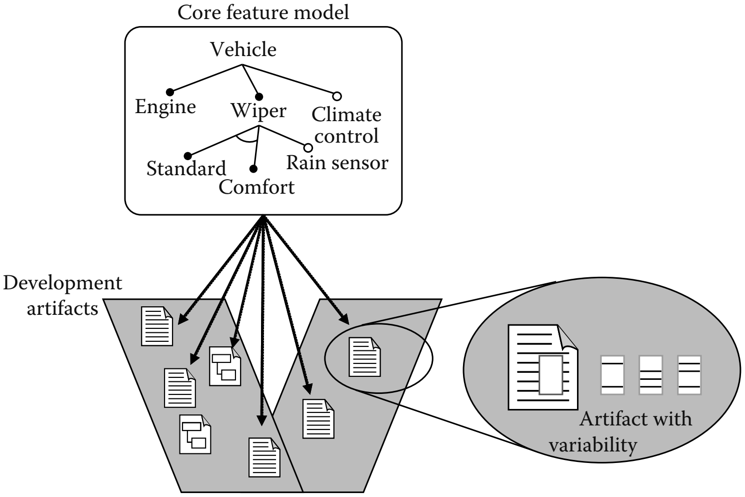


FIGURE 7.10 Feature-based configuration of artifacts.

- Due to the heterogeneity of different artifacts, for example, due to the variety of modeling and specification languages used in the automotive domain, it seems almost impossible to find a common variability mechanism that works well in all possible cases.
- However, even if artifact-specific mechanisms are used, still some connection has to be defined to the modeling of variability in the global coordination model.
- Moreover, in many areas, particular tools with proprietary modeling solutions and limited adaptation mechanisms have a dominant position in practice. Up to now, these tools do not provide comprehensive support for variability and product configuration, hence pragmatic solutions are developed in development projects. Unfortunately, this leads to a variety of different mechanisms to handle variability even when considering a single artifact and a single tool only.

As an example, consider the areas of ECU specification and of model-based analysis and development of ECU software. In these areas, very different artifacts have to be supported, that is, textual specifications and software models, and there are widely used tools with proprietary mechanisms, that is, DOORS and ML/SL/Stateflow (SF). Since support for variability is traditionally limited or not present at all in these legacy tools, a variety of solutions or workarounds have emerged in development projects. For example, because ML/SL/SF does not support explicit representation of variation points in its block models, two workarounds are commonly used in practice:

- One approach is to build a single model containing all variations in parallel and to configure individual variants by switching individual blocks on or off. The disadvantage is that such a model does not represent a meaningful system in itself and is thus not easy to comprehend and maintain.

- Another approach is to (ab)use an existing tool concept, for example, the concept of configurable subsystems in ML/SL/SF, to represent variability in a limited way.

A similar situation exists with respect to ECU specifications: widely used tools, such as DOORS do not offer concepts for explicit modeling of variation points and hence a variety of pragmatic solutions and workarounds are being used.

As a consequence of these considerations, we argue to handle artifact-local variability as follows:

- Since each artifact has its own characteristics and (pragmatic) constraints, different variability mechanisms must be supported for different artifacts.
- In order to connect these artifact-local variability mechanisms to the global coordination model, a common interface mechanism must be provided. The local feature models of artifact lines as described in Section 7.4.2 can serve as such an interface.
- In order to solve the problem of multiple representations of variability for a single artifact, a single intermediate (standard) representation is defined for each artifact on which all particular approaches can be mapped. This intermediate representation is then used as basis for connecting the artifact-local variability modeling to the global coordination model.

Due to the diversity of artifacts in the automotive domain, we cannot discuss and illustrate this approach for all types of artifacts here. Instead, in the remainder of this section, we will concentrate on a discussion of variability in ECU specifications. Using this as an example, we will demonstrate basic styles of representing artifact-level variability and illustrate important considerations to be taken into account when designing or selecting a certain form of representation, which are applicable to most other types of artifacts in a similar way.

7.5.3 Representing Variability in ECU Requirements Specifications

In order to represent ECU requirements specifications, practically all requirements management tools use a few common structures (Figure 7.11):

- Specification objects are hierarchically organized.
- On each hierarchical-level specification objects are sequentially ordered.
- Specification objects have predefined attributes, for example, the main specification text, as well as project-specific attributes such as the object's status.
- Specification objects are connected among itself and to other artifacts via traceability links.

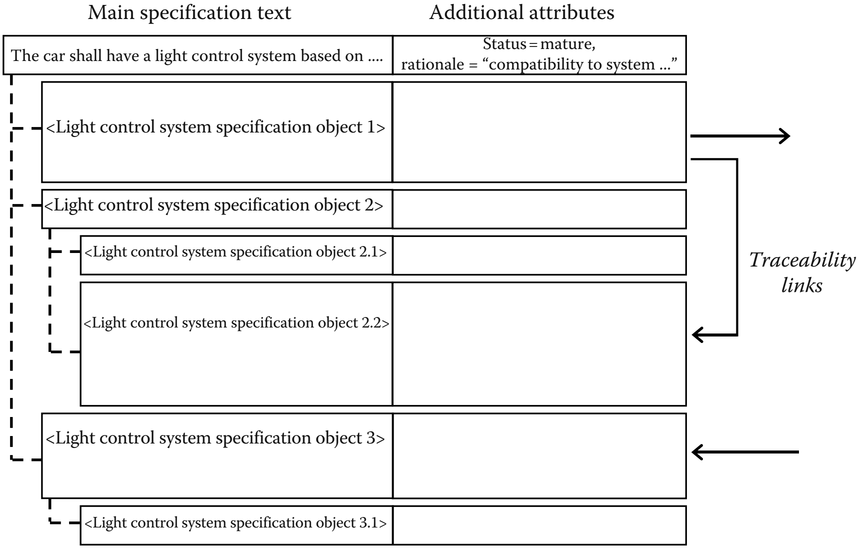

Main specification text
Additional attributes
The car shall have a light control system based on ....
Status = mature,
rationale = "compatibility to system ..."
<Light control system specification object 1>
<Light control system specification object 2>
<Light control system specification object 2.1>
<Light control system specification object 2.2>
<Light control system specification object 3>
<Light control system specification object 3.1>
Traceability links

FIGURE 7.11 Common structure of ECU requirements specifications.

Therefore, these structures are not tool-specific. In addition, they conform to the common requirements interchange format "RIF" [Her07], currently introduced by the German automotive manufacturers, and supported by several tool vendors.

Based on our experiences in different development projects, it turns out there are basically four approaches to represent variability in such specification structures.

1. *Using the object hierarchy to represent variability*: In this approach, a variation point is represented by a single new specification object. The child objects of this variation point then represent the specification parts belonging to this variation point. In the most simple case, the variation point models an optional specification part. This part is then directly placed hierarchically below the variation point. In more complex cases, the variation point has several child elements, that is, variants, and an additional hierarchical level is used to separate between these alternatives.
2. *Using the sequential ordering of specification objects to represent variability*: In this approach, a variation point is represented by two new specification objects (on the same hierarchy level), marking the beginning and the ending of the specification parts belonging to this variation point. As discussed above, additional intermediate objects may be used to distinguish between the variants of more complex variation points.
3. *Using traceability links between specification objects to represent variability*: In this case, a variability point is represented by a single object and all the specification elements belonging to this variation point are connected to this object via traceability links. Specific types of links or additional objects are used to represent more complex variation points.

4. *Using attributes of specification objects to represent variability*: In this case, no additional object is introduced to represent variation points, but a new attribute is introduced that specifies for each specification object to which variation point and to which variant thereof it belongs.

In Section 7.5.4, we briefly compare and evaluate these approaches.

7.5.4 Evaluation of Representations

We evaluate these approaches based on a few criteria that we consider relevant from practical experience in the field.

Clarity: The approach should not obscure the variability information within specifications.

Definition/maintenance effort: The approach should not lead to excessive definition or maintenance efforts.

Robustness with respect to hierarchy violations: When working on specifications, engineers often unintentionally violate the intended specification hierarchy, for example, when dragging and dropping objects during a reorganization of the specification. The approach should be robust with respect to such mistakes.

Robustness with respect to reordering: Similarly, engineers typically change the sequential order of specification objects, for example, when cleaning up specifications or simply to improve readability and understandability. The approach should be robust with respect to such changes too.

Table 7.1 summarizes a brief evaluation of the four approaches with respect to these criteria.

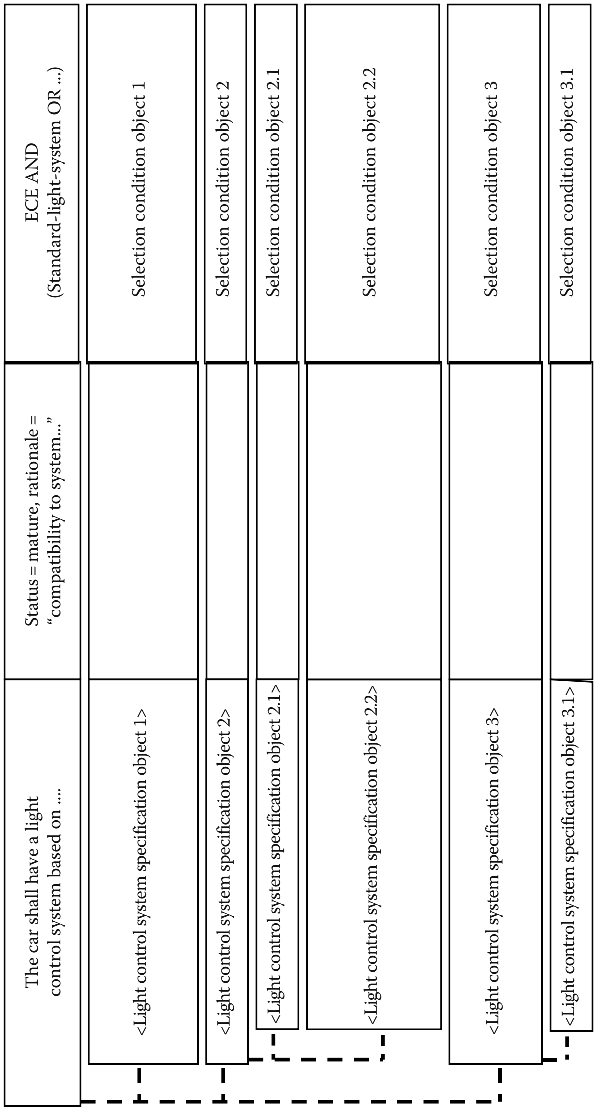
This evaluation shows that none of the mechanisms matches all needs: those approaches that are more robust with respect to specification reorganization tend to increase the effort and the obscurity, and vice versa. The decision for one of these approaches therefore needs to be taken on the level of individual projects while considering the project's specific characteristics and needs.

7.5.5 Mapping Representations to a Common Basis

The previous section has shown that none of the four representations is matching all needs. Hence it is likely that different mechanisms will continue to be used in industrial settings. In order to cope with this situation, a common representation may be used as a basis, to which all approaches can be mapped (Figure 7.12). The idea of this

TABLE 7.1 Evaluation of Various Form of Variability Representation in ECU Requirements Specifications

	(1) Use Object Hierarchy	(2) Use Object Sequence	(3) Use Links	(4) Use Attributes
Clarity	+	+	–	–
Definition/ maintenance effort	+	+	–	–
Robustness with respect to hierarchy violations	–	–	+	+
Robustness with respect to reordering	–	–	+	+



Main specification text	Additional attributes	Selection condition
The car shall have a light control system based on	Status = mature, rationale = "compatibility to system..."	ECE AND (Standard-light-system OR ...)
<Light control system specification object 1>		Selection condition object 1
<Light control system specification object 2>		Selection condition object 2
<Light control system specification object 2.1>		Selection condition object 2.1
<Light control system specification object 2.2>		Selection condition object 2.2
<Light control system specification object 3>		Selection condition object 3
<Light control system specification object 3.1>		Selection condition object 3.1

FIGURE 7.12 Representation of artifact-level variability with selection conditions.

intermediate representation is to implicitly consider all objects of the specification as variation points and to attach selection criteria to each object describing under which combination of feature selections this object is applicable.

Note that this approach can also be seen as an additional approach to represent variability, sharing the advantages and disadvantages of the attribute-based approach 4. It is relatively straightforward to transform any of the approaches 1–4 into this representation:

- In a first step, approaches 1–3 are transformed into the representation in approach 4.
- In a second step, the variation points used in the attributes are replaced by their selection conditions. Similarly, for individual variants of variation points.

References

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2002.

[Gri03] K. Grimm. Software technology in an automotive company—Major challenges. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, OR. Association for Computing and Machinery & IEEE Computer Society, 2003 pp. 498–503.

[Her07] Herstellerinitiative Software (HIS). *Requirements Interchange Format (RIF)—Specification*, May 2007. Version 1.1a.

[KCH+90] 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 CMU/SEI-90-TR-21, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, PA 1990.

[KKL+98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[MJA+04] D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Dörr, and K. Schmid. Gophone—A software product line in the mobile phone domain. IESE-Report 025.04/E, Fraunhofer IESE, March 2004.

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[PBvdL05] K. Pohl, G. Böckle, and van der F. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Heidelberg, 2005.

[RW05] M.-O. Reiser and M. Weber. Using product sets to define complex product decisions. In: *Proceedings of the 9th International Software Product Line Conference (SPLC 2005)*, Renner, France 2005.

[vO00] Rob van Ommering. Beyond product families: Building a product population? In: *Proceedings of the Third International Workshop on Software Architectures for Product Families (SAPF-3)*, Las Palmar de Gran Canaria, Spain LNCS 1951, 2000, pp. 187–198.

8

Reuse of Software in Automotive Electronics

Andreas Krüger
AUDI AG

Bernd Hardung*
AUDI AG

Thorsten Kölzow
AUDI AG

8.1 Reuse of Software: A Challenge for Automotive OEMs

At the start of the third millennium, the automotive industry is facing a new challenge. Ninety percent of all innovations are related to electronics, 80% of these are related to software. This means a big change for the development of electronics. More and more highly connected functions must be developed to series-production readiness, while at the same time development cycles become shorter and shorter. The importance of software in the automotive industry is shown very impressively in a study of Mercer Management Consulting and Hypovereinsbank [1]. According to this study, in the year 2010, 13% of the production costs of a vehicle will go in software (Figure 8.1).

With respect to these findings, the development process including software development methods for the automotive domain must be improved. Intensive work has

[*] Bernd Hardung is now with Elektrobit Automotive GmbH in Erlangen.

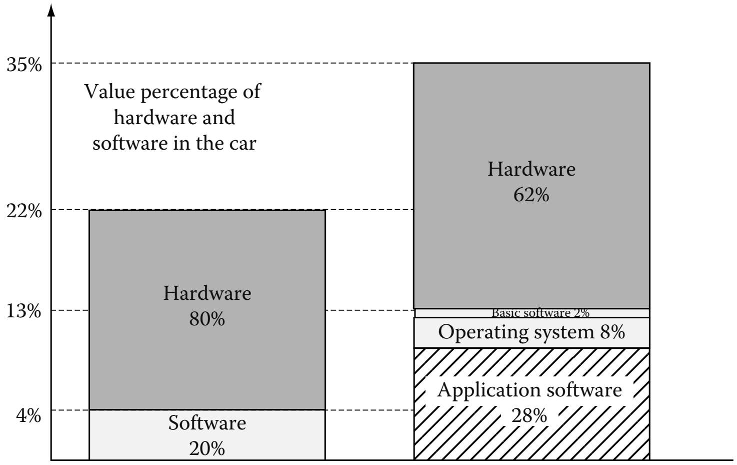


FIGURE 8.1 Rise of importance of software in the car. (From Mercer Management Consulting and Hypovereinsbank. Studie, Automobiltechnologie 2010. München, August 2001.)

been done in some parts of this field. Requirements engineering, software quality, or model-based software development are the most prominent examples. The goals are always to reduce software development time and to increase software quality. Another promising but nevertheless challenging way to reach these goals is the reuse of software. The main prerequisite for reusing software in the automotive domain is to separate the hardware of an electronic control unit (ECU) from the embedded software that runs on it.

Until recently, the automotive manufacturers' perception of ECUs was that of single units. They specified and ordered black boxes from their suppliers. After the delivery of samples, they also tested them as black boxes. For the automotive manufacturer, this procedure has the disadvantage that the software has to be newly developed for each new project, if the supplier is changed. This not only causes expenses, but also increases development time.

An automotive electronics supplier can usually concentrate his development efforts on his or her single part of the system. This is in contrast to the view of the automotive manufacturer, who is responsible for integrating the single parts to an entire electronics system. Networked units with distributed functions require the automotive manufacturers to have development processes and methods that allow to reuse software on the system level. Moreover, methods for the reuse of software enable manufacturers to develop their own competitively differentiating software, thus securing their intellectual property.

Following this introduction, recent work in this field is described introducing the electronic systems in a modern vehicle (Section 8.2). Then a framework that enables automotive manufacturers to reuse software is presented (Section 8.3). In Section 8.4 an example is given in which this framework is applied to a realistic automotive scenario. The chapter concludes with Section 8.5.

8.2 Requirements for the Reuse of Software in the Automotive Domain

To get an idea of the software reuse issue from the view of an automotive manufacturer, an introduction to the electronic systems in a modern car is given here. As mentioned before, innovation in the automotive domain is mainly driven by electronics. Improvements in fuel economy and engine power in the last years, but also driver assistance systems like "Audi Side Assist" (blind spot warning) and "Audi Lane Assist" (lane departure warning) are not imaginable without electronics.

In order to fulfill the increased communication needs of these electronic systems, the ECUs communicate via different bus systems. The most widely used automotive bus systems are controller area network (CAN) [2], local interconnect network (LIN) [3], and media-oriented system transport (MOST) [4]. FlexRay [5] is currently introduced in the first series cars. An example for the complexity of such a system is the network topology of the Audi A5 coupé, shown in Figure 8.2.

It was already pointed out in the introduction that car manufacturers increasingly begin to see software as a separate electronics component, independent from the underlying hardware. This view is a prerequisite to be able to reuse software, which, however, has a variety of further aspects and unresolved problems. In Ref. [6] requirements for the reuse of software within the automotive range are presented as follows:

- Reusable application software components must be "hardware independent."
- Interfaces of the software components must be able to exchange data both locally on an ECU and/or via a data bus.

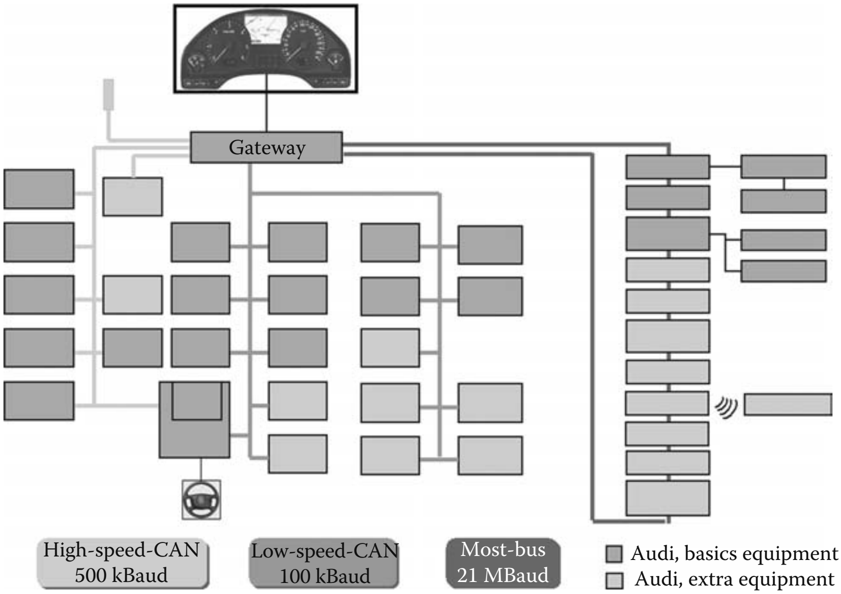


FIGURE 8.2 Electronic system of the AUDI A5 coupé.

- Developing reusable software, future requirements to each function must be considered.
- "Code size" and "execution time" of the software components must be minimized also when developing reusable software. Both resources are expensive due to mass production of parts in the automotive industry.

Additional aspects must be considered. Individual software modules must have an optimal "modularity." This means that one function (e.g., central door looking system or exterior light) might consist of different individual subcomponents. Building subcomponents improves the reusability, since a functional change can imply only changing a single subcomponent.

The partitioning of functionality into subcomponents however can cause repetitions of code. For example, multiple variable declarations lead to a higher memory consumption of all submodules together in comparison to a module developed as single unit. Moreover, execution time might worsen.

The interface definition of the software modules must be specified once, that means statically. It should not change during the reuse in a new car model or on a new microcontroller. The interfaces must be maintained in a database over all model ranges.

A further requirement is the existence of a database in which the individual reusable software components and/or subcomponents are stored. A "uniform data format" is required to exchange data between different software development teams at the car manufacturers and their tier 1 and software suppliers. In order to fill and use the information and the reusable software components from the database, "processes" must be defined that enable a standard development process. In particular, these processes must describe the integration process for software from different sources and a role model including both manufacturer and suppliers.

To support these processes, we need a seamless "tool chain," that is, a suite of tools cooperating closely via tightly matching interfaces. An important aspect thereby is the use of "uniform modeling guidelines." Likewise, standards should be defined between suppliers and manufacturers to effectively facilitate the exchangeability of software modules. From the view of a car manufacturer, there exist "different kinds of reuse." According to the kind of reuse of a software component—on the same model or over different model ranges—different aspects must be considered.

Of course, the safety aspect must be taken into consideration. In Ref. [7] requirements for the development of safety critical functions are stated. Simonot-Lion gives an overview on several aspects, for example, verification process, time-triggered architectures, and software architecture models.

8.3 Supporting the Reuse of Application Software Components in Cars

The Software Engineering Institute of the Carnegie Mellon University in Pittsburgh developed a process model named "product line practice" (PLP) [8]. We propose

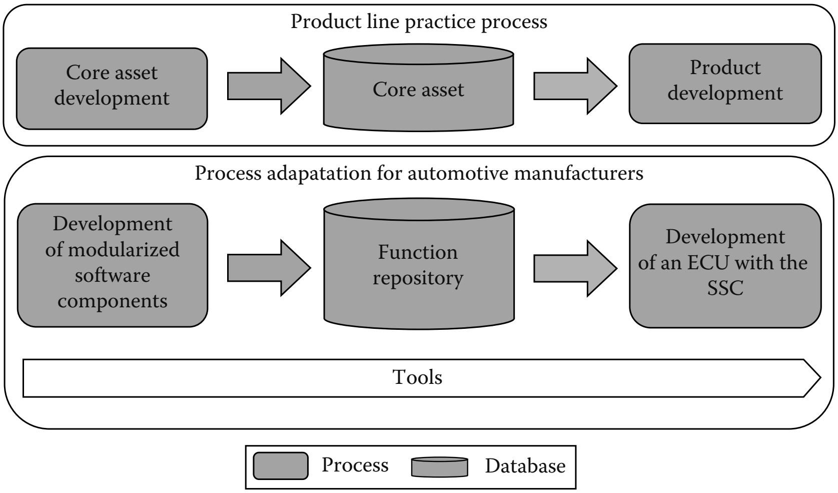

Product line practice process
Core asset development
Core asset
Product development
Process adapatation for automotive manufacturers
Development of modularized software components
Function repository
Development of an ECU with the SSC
Tools
Process
Database

FIGURE 8.3 Framework components of the automotive PLP application.

a framework based on this approach for the reuse of application software in the automotive domain.

The term "product line" (Chapter 7) is thereby defined as follows [9]:

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of *core assets* in a prescribed way."

The terms and the processing model of the PLP are applied on software reuse by car manufacturers. Figure 8.3 shows the components of the presented framework. The top part of the figure shows the generic PLP method, whereas the bottom part depicts the instantiation of the PLP method for automotive manufacturers, including the necessary tools.

The section first explains the general processes of the PLP (Section 8.3.1). This is followed by a discussion how to modularize what are called the core assets (Section 8.3.2). Thereby the core assets are stored in a database called "function repository" (Section 8.3.3). Further, we describe how to develop products with the content of the function repository as building blocks and using a standard software core (SSC) (Section 8.3.4). The tools required to support the process are described in Section 8.3.4.2.

8.3.1 Processes

According to Ref. [8] the process in a product line is divided into three different areas:

- *Core asset development*: When developing core assets, first a list of the products that are desirable from today's point of view is created. This list is defined as the "product scope." Thus, it contains also products that may be realized in the future and that are not a goal of the current development. The list of products represents a boundary, which should be thought over very carefully: If the scope of production is too wide, many of the core assets can be used only once. In this case, no advantage compared to conventional development is achieved. If the scope of production is too narrow, the future variety of the product is limited unnecessarily.

- *Product development*: In addition to the product scope and the core assets, there are also product-specific requirements. With the core assets, the development of a new product within the product scope is equal to combining some of the core assets. The description of the process of combination is called the "production plan." The production plan is a general description and the product development should fulfill the product-specific requirements. Depending on the accuracy of the production plan the product must be developed under consideration of "variation points." In Figure 8.4 [8], core assets are shown as rectangles and the correspondent processes as triangles. Put together these processes result in the production plan, from which the product can be developed.
- *Management*: The management is divided into a technical and an organizational part. The organizational management must provide the right form of organization and the needed resources (this includes also the training of the employees). The technical management is responsible for realizing the core asset development and the product development.

The engineering tasks of the processes within the presented framework on the one hand contain producing and archiving reusable software components into a database. On the other hand, these software components must be used to develop new products.

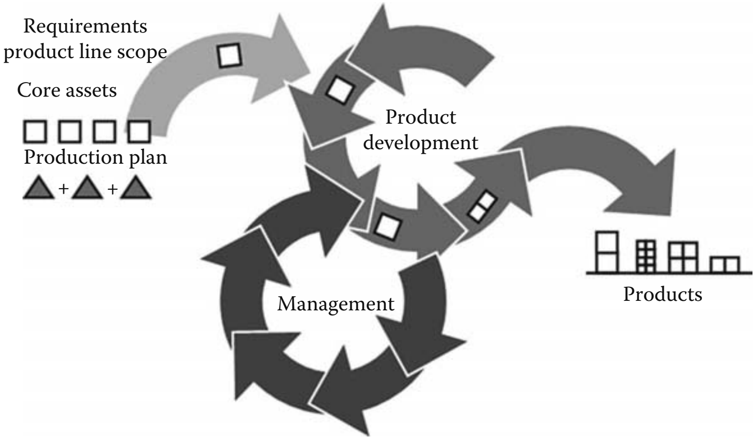


FIGURE 8.4 Product development in the PLP. (From Carnegie Mellon Software Engineering Institute. With permission.)

In Ref. [10], a practical application example of the automotive domain is discussed and a method is described that allows controlling the variability of a product within the product development process.

As in-vehicle embedded applications are largely distributed and networked, we present an approach that explicitly considers these properties and furthermore focuses on modularization of the functional software and the relation to an SSC.

8.3.2 Development of Modularized Automotive Software Components

Figure 8.5 shows the process for developing the complete electronics system of a new model, and the circular dependency between its stages. With "functional architecture" we denote the definition and allocation of functional (software) modules to ECUs. This allocation results in communication relationships that define the signals and messages exchanged via the in-car bus systems (definition of the communication). The "hardware system" consists of ECUs with their central microcontrollers, which need to have adequate computing power, on-chip peripherals (e.g., communication controllers, interfaces to sensors and actuators), and further electronic components.

Thereby, the three main process steps must be conducted in an iterative way, since each step depends on the other. Therefore, the allocation of software modules to ECUs must take place after defining the hardware system. However, this allocation depends on the memory and processing power of the ECUs. Of equal importance is the question whether the bandwidth limitations of the bus system allow to transfer all necessary signals between the individual software modules.

This process requires modularized software components to separate hardware-dependent and hardware-independent parts. The modularization of software components according to definitions of movability and reusability also enables reuse in this particular process. To support the classification of software according to their movability and reusability, the following terms shall be introduced: "firmware," "basic software," "adaptation software," and "function software."

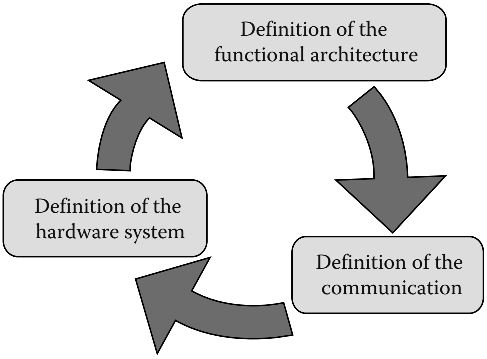

Definition of the functional architecture
Definition of the communication
Definition of the hardware system

FIGURE 8.5 Product development process for an automotive electronics system.

- *Firmware* is the hardware-dependent part of the ECU software. Examples are the communication drivers, drivers for analog–digital converters, or drivers for pulse width modulators.
- *Basic software* is the software that is independent from application and hardware. Examples are communication interaction layers between application software modules and communication drivers. They are not dependent on the hardware since they use the application programming interface (API) of the communication drivers (see also Section 8.3.4).

The two categories basic software and firmware are reused already today. They are called "standard software," although the partitioning of the components is not only made with respect to the reusability (Section 8.3.4). Another reason for decomposing the standard software is the possibility to fulfill the needs of different classes of ECUs by removing standard software components that are not needed.

We define the two remaining categories of software as follows:

- *Adaptation software* is the application-specific part of the software that adapts the function software to the car model and builds the connection from the function software to the firmware and the basic software.
- *Function software* is the function-specific part of the software that is independent from the car model it is used in.

With these new terms, the types of components can be separated according to the type of movability and reuse. Considering this abstract model three types of reuse can be defined:

- Reuse over different ECUs
- Reuse over different microcontroller platforms
- Reuse over different model ranges

The classification of the types of software and the corresponding types of reuse are shown in Table 8.1. This classification gives guidance as to how the modularization can be performed in order to get the maximum movability and reuse effect. For example, a communication driver (firmware) can be reused over different ECUs and over different model ranges (provided that they use microcontrollers of the same family), but not over different microcontroller platforms, since the communication controllers usually are quite dissimilar on different microcontrollers. On the other hand, function software that is based on the proposed software layers can be reused across ECUs, microcontrollers, and model ranges.

TABLE 8.1 Types of Software with Respect to Reuse

Reuse over Different	ECUs	μC-Platforms	Model Ranges
Firmware	Yes	No	Yes
Basic software	Yes	Yes	Yes
Adaptation software	No	Yes	No
Function software	Yes	Yes	Yes

8.3.3 Function Repository

The function repository contains the core assets (Figure 8.3). The question what the core assets are is essential for the reuse of software. Reference [8] discusses in general what items build the core assets. In the following, the set of core assets for a function repository of an automotive manufacturer is introduced, thereby considering the special requirements as described in Section 8.2. The function repository must not only contain descriptions of the reusable software. Hardware and bus system descriptions are also an essential part.

- *Software components*: An important content of the function repository are the reusable software components. Here, a distinction can be made according to different categories of software and different programming languages they are implemented with.

  Categories of software are firmware, basic software, adaptation software, and function software as described in Section 8.3.2. Additionally, the characteristics of the software components must be stored in the function repository. For example, code size and worst-case execution time of the individual software components and subsystems are important information for a proper real-time integration into an operating system. The worst-case execution times must be determined for each supported hardware platform of the software component. This must be done, for example, with model-based automatic approaches like in Ref. [11]. Another possibility is to obtain this information during the integration process. This procedure has the advantage that the environment can be taken into account. For example, in a certain car model some function is never used, the code is optimized and the worst-case execution time is reduced. This approach in general will derive worst-case execution times, which are tighter to the real execution time.

  In order to further improve the software reuse specifications, test plans (what to test when with which test equipment) and test cases (description of how to test individual properties of the device under test) should be stored as well (see also Chapter 11).

- *Interfaces*: The interfaces between the software components must also be part of the function repository. In an automotive system, communication between software components can be realized via a data bus or internally on the ECU. Interfaces between software components must be defined globally. New software components must use already available interface definitions. This ensures interface compatibility for different software components.

  The number of interfaces that must be described is enormous in a modern vehicle. To be able to manage the complexity it is necessary to specify interface types, of which interface instances can be derived.

  The description of interface types must not merely contain the name and bit size of the signals exchanged through the interface. Further essential information that must be defined is the supported communication type(s), which allow the system designer to correctly map the signals on communication busses. Once the interface instances are mapped to software components, timing requirements must be specified, which also influence the decision whether a certain communication bus system is suitable.

The semantics of the interface types is another important item. For example, once a user presses the remote key, the signal described by the type "remote key" is ON as long as the key is pressed, but at least 50 ms. The description of these semantics requires a formal method in order to verify the properties of interoperability between components (type interoperability, timing interoperability, etc.).

- *Functional network*: The functional network combines the software components to an overall software system that performs the car's functions. Here, the function repository ideally supports hierarchical decomposition in order to reduce the complexity that must be managed.

  An add-on for the interface definition of software components is a so-called error matrix. This error matrix can help when tracing errors in the automotive system after production in the field. The error matrix contains information as to which output of a software component depends on which input. With that, it is easier for the service to find software errors. A more detailed description can be found in Ref. [12].

- *Hardware platform and bus system description*: The entire software contained in the function repository is not independent of the hardware that it should run on. The execution of software depends on the type of processor, the amount of memory, the clock cycle or even the compiler that is used. Therefore, a "hardware platform description" must be stored in the function repository.

  As mentioned before, parts of the communication between the software components in functional networks use bus systems. These bus systems can be distinguished in different ways, for example, speed of data transfer, used protocol, and so on. Thus for the development of a new car model it is also necessary to store the relevant data in the "bus system description" of a function repository.

- *Implementation*: For the implementation (or "behavior description") of the software components, model-based methods and tools can be used. From these models, usually C code is derived. Generated code, however, might show performance disadvantages when compared to handwritten C code. The advantage of the model-based software components, on the other hand, is that they can be adapted easily to different hardware platforms. Therefore, it might seem useful to store both variants—handwritten and generated code—in the function repository depending on the kind of software.

8.3.3.1 Standards for the Storage of Data

All the mentioned data for the function repository must be stored in a database. We suggest to use a standardized data model for this purpose.

Within the EAST-EEA project [13] (www.east-eea.net, see also Chapter 9) an architecture description language (ADL) [14] was developed in order to be able to specify functions without considering the hardware. An ADL allows the description of the structure (but not necessarily the behavior) of software in a functional approach as described in this section.

A further example for such a data model is MSR MEDOC [15,16]. It defines how the data has to be stored in the data model. The data model is specified in an eXtensible Markup Language data type definition (XML DTD).

Another important project is automotive open system architecture (AUTOSAR) [17]. This project defined a modular software architecture. Here, the important aspect of integrating software components from different software suppliers is considered as well.

8.3.4 Development of an In-Vehicle Embedded System

8.3.4.1 Automotive Standard Software Core

For product development based on the modularized software components and the other stored assets, an automotive SSC is used. Standard software comprises all tasks that can be standardized, that is, that are to some extent independent from a specific application. These tasks are, for example, controlling the hardware drivers, recognizing and storing errors, and controlling the network connection. These functions are implemented as separate, reusable modules. The sum of all standard software components is the SSC.

The SSC is the part of the ECU software, where reuse is already performed today—even if the ECU supplier changes. The degree of reusability of a software component depends on the software category as defined in Section 8.3.2.

Figure 8.6 shows the structure of the SSC of the Volkswagen group. First of all, it consists of an operating system conforming to the OSEK standard [18].

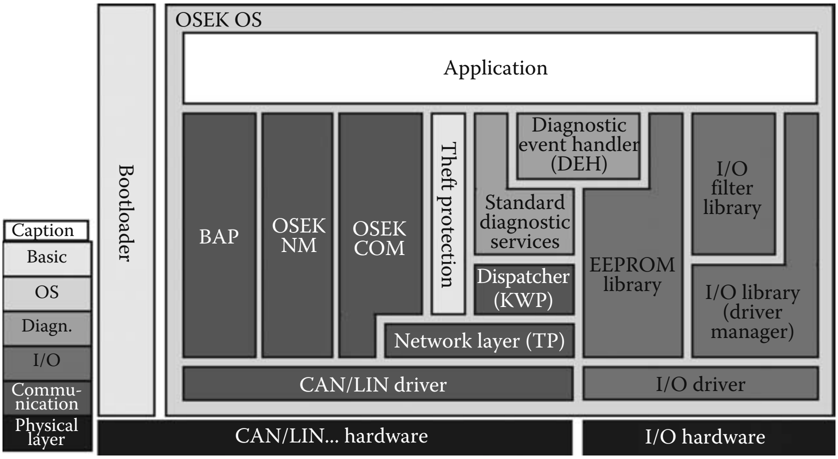


FIGURE 8.6 Example of an SSC: the Volkswagen group SSC.

Communication drivers (CAN and LIN) are the basis for higher-level protocol modules: network management (NM), control and display protocol (BAP, German for Bedien- und Anzeigeprotokoll), transport protocol (TP) or the signal-filtering layer OSEK COM [19]. Diagnosis functions are supported by the modules "diagnostic event handler" and "standard diagnostic services." Basic hardware abstraction is implemented by the input/output (I/O) driver and library modules, which handle access to EEPROM, digital–analog converters, analog–digital converters, pulse-width modulators, external ports, etc. Finally, a separate bootloader software allows to program each ECU with flash memory via the car's diagnosis interface. A more detailed explanation can be found in Ref. [20].

The SSC not only supports a hardware-independent interface, but also the entire infrastructure services on the microcontroller that are used by the actual application. This facilitates an application development that is independent from the used microcontroller platform.

The interface between SSC and the function software is typically realized through APIs. A good example of an API is the OSEK COM [19] specification. The function repository as described contains all necessary information to configure the APIs of standard software modules. Thus, the combination of standard software modules represents the basis for an efficient reuse of the application as already explained in Section 8.3.2.

Since the architecture of the SSC is not only designed with respect to reuse aspects, the reuse effect can still be improved. Nevertheless, standard software plays an important role for reusing function software. For this reason, an industry cooperation called AUTOSAR was initiated in 2002 with the objective to further develop and standardize already existing standard software architectures [21]. The AUTOSAR specifications focus on the description of interfaces between software components and of functions performed by these components. They do not specify particular implementations, since this is left to the software suppliers providing AUTOSAR-compatible standard software.

Figure 8.7 gives an overview of the AUTOSAR software components and interfaces. The central element is the AUTOSAR runtime environment (RTE), which provides a communication abstraction (both inter- and intra-ECU) to the application software. In our terminology, the RTE is a typical piece of adaptation software. Beneath the RTE the figure shows a vertical distinction into system services (e.g., timers and counters), memory services (e.g., access to EEPROM), communication services (e.g., packing an application signal into a CAN message and managing the transmission in a timely manner), I/O access (e.g., handling analog–digital conversion), and so-called complex device drivers (e.g., handling application-specific peripherals that need direct support by the ECU hardware and cannot be standardized). Another, horizontal distinction is made between services, abstraction layers, and drivers.

Further information on AUTOSAR (Chapter 2) can be found in Ref. [21] and on the AUTOSAR Web site [17].

8.3.4.2 Tools Required for the Development Process

Obviously, tool support is required for all processes described so far. In a typical development environment, the goal of tool users is to obtain a seamless tool chain by

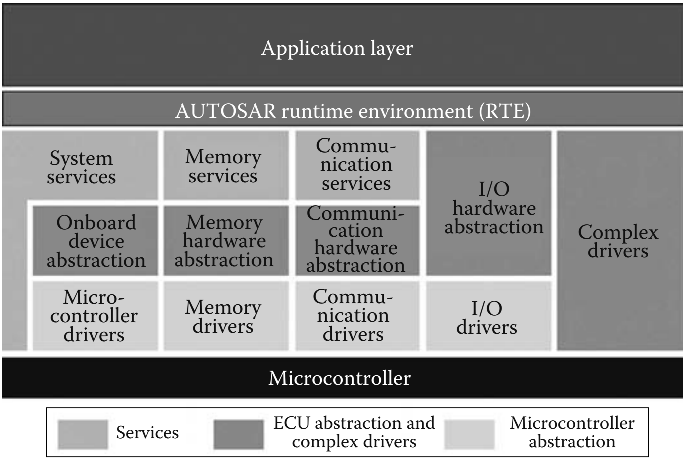


FIGURE 8.7 AUTOSAR software architecture.

developing conversion filters between the tools. This approach does not work when introducing a function repository. For every pair of tools used in a company or department such a filter would have to be developed, which translates the data from the database to the tool and vice versa. Since the language coverage is different for every tool, there is also some loss of information with every conversion. That is no issue as long as going along the V model [22] in one direction only. As soon as the development takes place in different locations of the V model at the same time (simultaneous engineering), the data cannot be kept consistent automatically anymore.

The solution for this problem is a standard data model as required in Section 8.3.3. The tools are then the editors for the data in the function repository. With this understanding a tool chain is a set of tools working on a common database, without manual user interaction.

8.3.4.2.1 Core Asset Development

The tools needed for core asset development must support different levels of system descriptions:

- *Requirements* are the textual description of what the user expects from the vehicle. In order to manage the requirements it makes sense to use tools like DOORS [23], which provide features like automatic requirement key generation, an extended search engine, and document generation possibilities.
- *Interfaces* are all points within an application or module where information flows into or out of the respective module. A tool is needed to manage these external interfaces. Also internal data might be interesting for debugging purposes. With the Data Dictionary [24], it is possible to manage external interfaces and internal data in one tool.

- *System architecture* is the functional network, the hardware architecture, and the mapping between hardware components and functional network. The system architecture can be described for example, with unified modeling language (UML) tools or automotive-specific tools like DaVinci [6].
- The *behavior description* can be either C code or models in various modeling languages like UML. Another option very common for the automotive industry is the use of Matlab/Simulink/Stateflow [25]. A tool like Targetlink [26] could be used for generating C code out of the model even if the target platform does not support floating point code.

8.3.4.2.2 Tools Required for Product Development

Most of the standard software components like network drivers for the CAN bus [27] or the OSEK operating system [18] must be adapted to the specific network node and hardware platform. This is not possible without adequate tool support.

A tool supporting such a component configuration needs information about the communication between the several control units. Each component requires specific data that are important for the function of the component. The variety of today's configuration tools is due to different data formats that the tools are based on. For the system integrator as the only one who knows about the overall communication between the control units, this means a great effort in handling the several data formats and a potential risk of introducing errors. Therefore, a single database for the communication information, like in Section 8.3.3 must be created.

This adaptation is done by configuration tools with integrated code generators. Here a weakness of today's SSCs can be seen. There is no unique configuration interface for the whole SSC, but many component there are specific configuration tools. As a result, it is only possible to optimize the software components, but there is no tool support for optimizing the overall system. This optimization depends on the experience and the knowledge of the system developer. Nevertheless, there are already ongoing efforts to integrate the several tools in an open framework with one unified user interface.

8.4 Application Example

In this section, an example is given as to how software modularization facilitates the reuse of software components. Some highlights of the process of modeling the function architecture and implementing it on hardware shall be demonstrated using the function exterior light. This function was chosen because it is quite challenging to reuse this function over different car models without changing the interfaces.

The requirements of this function are derived from existing specifications. Apparently, there are differences in the partitioning of the rear light of cars. This is even true for different car models of a single car manufacturer. Figure 8.8 shows the expected behavior of the rear lights when combining the on/off states with the brake light on/off states. We further assume that one car model has one bulb while another has two bulbs

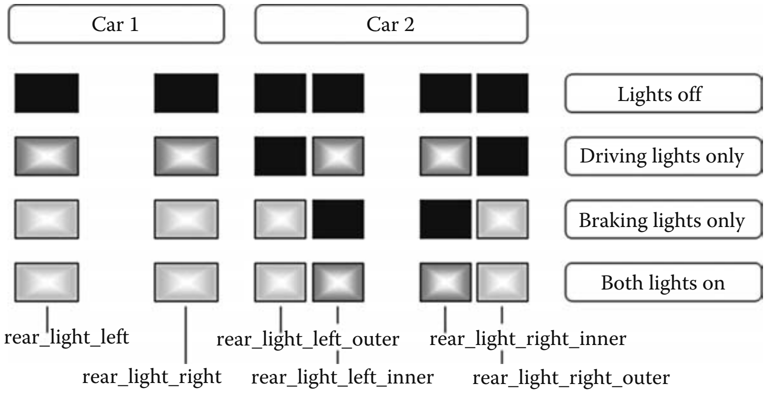


FIGURE 8.8 Example for different arrangements of rear lights.

per side. Further, there is a different level of brightness when the driving light is on and when braking.

The application exterior light controls the bulbs via interfaces of the SSC. For the car with two bulbs per side these are "rear_light_left_outer," "rear_light_left_inner," "rear_light_right_inner," and "rear_light_ right_outer." For the car with one bulb per side the interfaces are "rear_light_left" and "rear_light_right." In addition, the third brake light must be handled. The underlying layers also support the input interfaces. Some of them are network interfaces to other ECUs, while some are internal interfaces within the ECU where the exterior light function is located. In Figure 8.9, two nonreusable software components can be seen. Both are implemented as monolithic modules, so that reuse is prohibited. The hardware is directly reflected in the output signals of the module controlling the rear lights, so that the modules cannot be reused when using the same input signals for other rear light arrangements.

In Figure 8.10, the software component implementing the light control logic is extracted (exterior light common). Adaptation software components, which are different for each car model, are used as interfaces. Two separate signals, brake_light and driving_light are introduced. These signals are output signals of the new generic

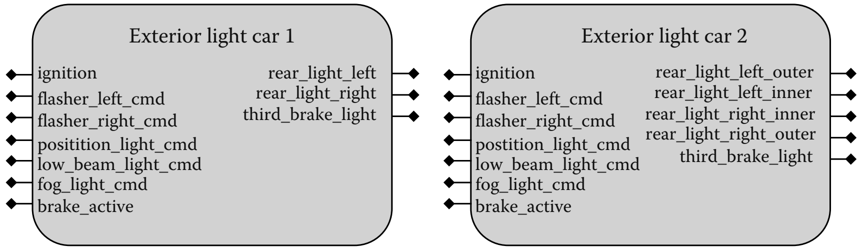


FIGURE 8.9 Not reusable software components.

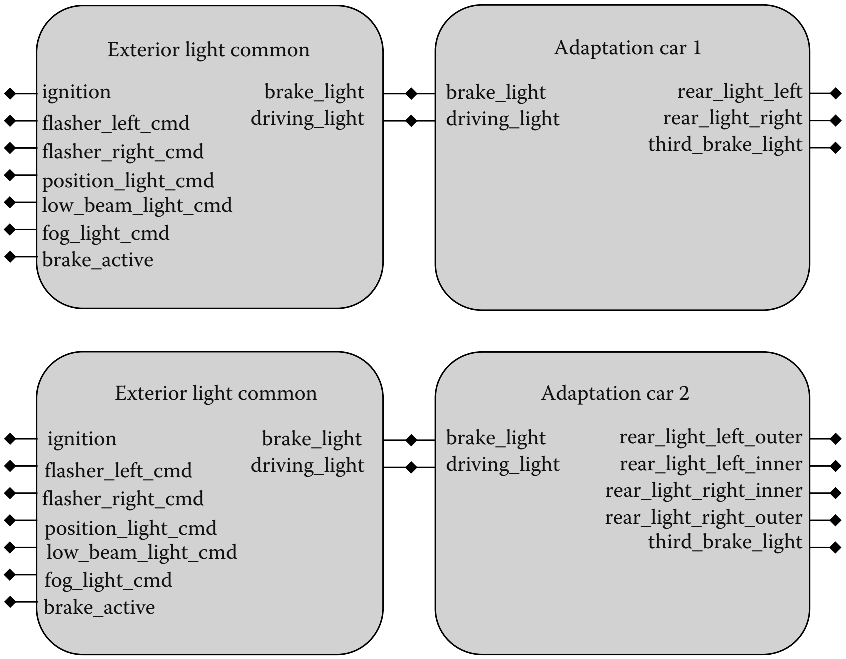


FIGURE 8.10 Reusable application software component.

module "exterior light common," and input signals for the adaptation modules needed for each exterior light arrangement.

The underlying hardware must also be considered. Figure 8.11 shows that the hardware can differ significantly in different scenarios. In scenario 1 shown on the left-hand

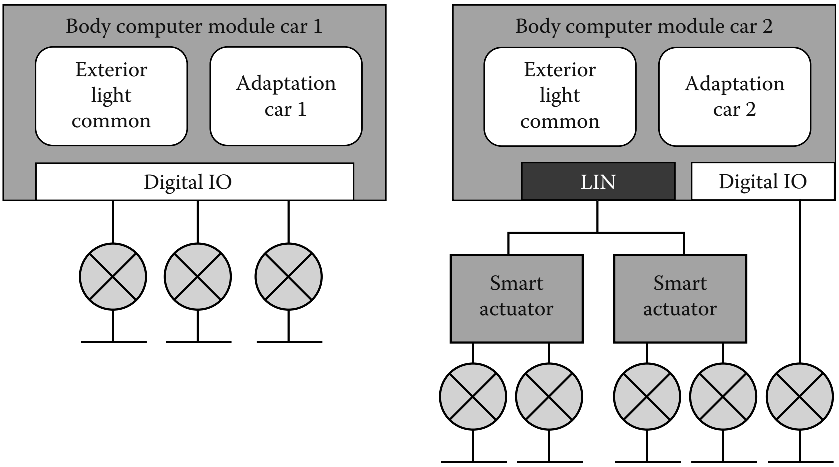


FIGURE 8.11 Different hardware scenarios.

side, the bulbs are actuated directly by digital input and output ports. In scenario 2 on the right-hand side of the figure, the third brake light is still handled by a discrete wire. The rear lights are so-called smart actuators. The main unit controls the smart actuators via a LIN bus in order to reduce the wiring harness.

Admittedly, this example is a very simple one. Nevertheless, we believe that it at least shows the principle and that the method is definitely applicable to more complex applications.

To get an impression of the real complexity of the systems handled in the automotive industry another real-life example, namely controlling the backup light, shall be explained shortly. In Figure 8.12, one can see that nine ECUs are involved to determine whether the backup lights should be switched on and which other actions must be performed when the driver selects the reverse gear.

First, the gearbox transmits the information that the driver has selected the reverse gear. In all Audi cars, this information must be routed through the central gateway. One of the body control modules (body control ECU front) hosts the entire exterior light control. It decides that the backup lights must be switched on and transmits this information to another body control ECU (rear), which actually switches on the backup lights. The same information is used by the door control modules, which dip the mirror on the passenger side if the driver has enabled this feature via the control panel in the driver's door. At the same time, the roof module must undip the automatic rearview mirror. Finally, if the car has a trailer, the backup lights of the trailer must be switched on via the trailer ECU, and the Audi parking system must switch off the rear sensors, in order not to confuse the driver with a permanent warning tone.

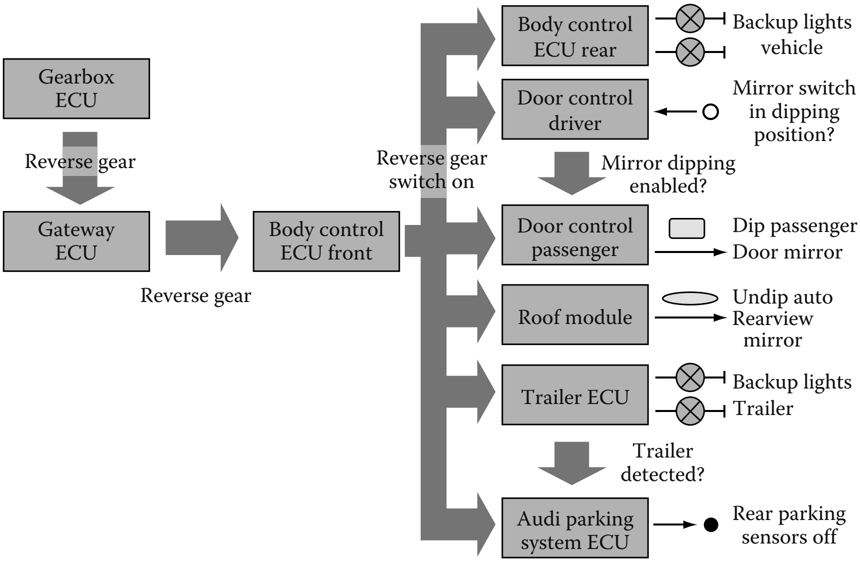


FIGURE 8.12 Example reverse gear.

8.5 Conclusion

This chapter gave an overview on the challenge of reusing software in the automotive domain from the perspective of an automotive manufacturer.

We explained the challenge of reuse in the automotive environment and pointed out the different notions of reuse at a tier 1 supplier and at a car manufacturer. The PLP process model was used to derive the requirements for reusing function software. Moreover, a method to classify software according to its potential categories of reuse was introduced. It was argued that this is a prerequisite to transform the current process of ordering black box control units into a process where software components can be reused even after exchanging suppliers. In conclusion, an application example was shown applying some parts of the presented framework.

References

1. Mercer Management Consulting and Hypovereinsbank. Studie, Automobiltechnologie 2010. München, August 2001.
2. CAN in Automation. Controller Area Network (CAN)—an overview. http:// www.can cia.de/can/, March 2003.
3. LIN Consortium. LIN Specification Package Revision 2.0. http://www.lin subbus.org, September 2003.
4. MOST Cooperation. MOST Specification Rev., 2.2. http://www.mostnet.de/down loads/Specifications/, November 2002.
5. H. Heinecke, A. Schedl, J. Berwanger, M. Peller, V. Nieten, R. Belschner, B. Hedenetz, P. Lohrmann, and C. Bracklo. FlexRay—ein Kommunikationssystem für das Automobil der Zukunft. *Elektronik Automotive*, September 2002, pp. 36–45.
6. B. Hardung, M. Wernicke, A. Krüger, G. Wagner, and F. Wohlgemuth. Development Process for Networked Electronic Systems. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles*, Baden-Baden, Germany, September 2003, pp. 77–97.
7. F. Simonot-Lion. In car embedded electronic architectures: How to ensure their safety. *Fifth IFAC International Conference on Fieldbus Systems and Their Applications—FeT'2003*, Aveiro, Portugal, July 2003, pp. 1–8.
8. Carnegie Mellon Software Engineering Institute. Software Product Lines. http://www.sei.cmu.edu/plp/product_line_overview.html, February 2003.
9. P. Clemens and L. Northop. *Software Product Lines—Practices and Patterns*. Addison-Wesley, Boston, MA, 2002.
10. S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, July 2002, pp. 66–72.
11. R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink Models. *Proceedings 14th Euromicro International Conference on Real-Time Systems*, Vienna, Austria, June 2002, pp. 31–40.
12. J. Schuller and M. Haneberg. Funktionale Analyse—Eine Methode für den Entwurf hochvernetzter Systeme. *Vortrag, VDI-Mechatronik-Konferenz*, Fulda, May 2003.
13. EAST-EEA. Embedded Electronic Architecture. http://www.east-eea.net, April 2004.

14. T. Thurner, J. Eisenmann, U. Freund, R. Geiger, M. Haneberg, U. Virnich, and S. Voget. The EAST-EEA project—a middleware based software architecture for networked electronic control units in vehicles. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles*, Baden-Baden, Germany, September 2003, pp. 545–563.
15. J. Hartmann, S. Huang, and S. Tilley. Documenting software systems with views II: An integrated approach based on XML. *Proceedings of the 19th Annual International Conference on Computer Documentation*, Santa Fe, New Mexico, October 2001, pp. 237–246.
16. MSR. Development of Methods, Definition of Standards, Subsequent Implementation. http://www.msr-wg.de, April 2004.
17. AUTOSAR. Automotive Open System Architecture. http://www.autosar.org, January 2007.
18. OSEK/VDX. Operating System Version 2.2.1. http://www.osek-vdx.org/mirror/os221.pdf, 2003.
19. OSEK/VDX. Communication Version 3.0.1. http://www.osek-vdx.org/mirror/com301.pdf, 2003.
20. A. Krüger, G. Wagner, N. Ehmke, and S. Prokop. Economic considerations and business models for automotive standard software components. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles*, Baden-Baden, Germany, September 2003, pp. 1057–1071.
21. H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürst, K.-P. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Voget, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and exploitation of the AUTOSAR development partnership. *SAE Convergence Conference*, Detroit, MI, Paper No. 2006-21-0019, October 2006.
22. Bundesministerium des Inneren. Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell. Kurzbeschreibung, Bonn, 1997.
23. Telelogic. Telelogic Doors Overview. http://www.telelogic.com/products/doorsers/doors/, April 2004.
24. C. Raith, F. Gesele, W. Dick, and M. Miegler. Audi Dynamic steering as an example of distributed joint development. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles*, Baden-Baden, Germany, September 2003, pp. 185–205.
25. MathWorks. MATLAB and Simulink for Technical Computing. http://www.mathworks.com, April 2004.
26. DSpace. TargetLink—Automatic Production Code Generation for Target Implementation. http://www.dspace.deU/ww/en/pub/products/targetimp.htm, April 2004.
27. HIS—Hersteller Initiative Software, Volkswagen AG. HIS/Vector CAN-Driver Specification V1.0. http://www.automotive-his.de, August 2003.

9

Automotive Architecture Description Languages

Henrik Lönn
Volvo Technology Corporation

Ulrich Freund
ETAS

9.1 Introduction

Developing automotive electronics is an increasingly challenging task, due to the increased complexity and expectations of the vehicle systems, the electronics and software architectures, and the development processes. This chapter addresses architecture description languages (ADLs) as a means to manage the engineering information related to automotive electronics.

For several reasons, the electronics and software content of the vehicle has grown rapidly over the years. Pollution legislation initiated the trend by its requirement on refined engine control, which could only be accomplished by computer control. This was an example of a stand-alone computerized system that was replacing an existing system in a way that was transparent to the driver. Since then vehicle electronics has evolved toward providing added value by new features, and vehicle control systems interact to optimize performance and functionality. Along the way, a vast majority of the vehicle's functionality and character has become affected by the electronic systems. As a result, the complexity and criticality of vehicle electronics is now highly challenging.

The bottleneck is no longer the technology, but the engineering capability. In particular, managing the engineering information is critical.

The chapter starts with a characterization of the needs regarding engineering information management, followed by a description of current states of practice. It then presents the potential of using an ADL and discusses different alternative approaches. The chapter closes with a conclusions section.

9.2 Engineering Information Challenges

An ADL for automotive electronics needs to meet the needs for information exchange and capture engineering information of various kinds. Below we discuss the needs and challenges that apply.

9.2.1 Reducing Cost and Lead Time

Cost reduction and efficiency in development are of course the root challenge, from which most of the other aspects are derived. Product development cycles are under high pressure. For example, Volvo cars has introduced around 15 new models in the last 10 years, compared to half of that in the previous decade. Other manufacturers may have an even larger increase. Maintaining high quality under these constraints requires advanced methodology and information management.

9.2.2 Development Organization and Information Exchange

A vehicle is developed in a complex organizational setting, where the engineering information needs to cross several interfaces. First, the vehicle manufacturer has a multitude of departments. The prime stakeholders are the engineering departments, but these need to interact with, for example, marketing and after sales. The engineering departments are organized according to geographical locations, product lines, vehicle domains, and work content, and thus represent a complex grid of information exchange.

Second, the vehicle original equipment manufacturers (OEMs) rely on their first-tier suppliers for the design, implementation, and manufacturing of components. This means that large amounts of product documentation needs to be exchanged between the supplier and the OEM. The supplier has a large internal organization with similar information exchange processes as the OEM.

Third, first-tier suppliers have additional suppliers where engineering information have to be exchanged in the same way, although the domain-specific content is reduced for second- and third-tier suppliers.

9.2.3 Product Complexity

The increased complexity of the product stems from the complexity of the individual functions, and the increased interaction between functions. The increased interaction is true for most functions, although the active safety domain is a particularly large contributor. Several chassis, telematics, and human–machine interface (HMI) functions are integrated to provide warnings, assistance, and mitigation.

The advances in system and software architecture, where several applications are integrated in the same control unit is another trend that adds to the complexity. On the one hand, componentization, which is an important part of the trend, makes interfaces and resource requirements more well defined. On the other hand, issues like integrity and safety require new mechanisms that add to the complexity.

9.2.4 Quality and Safety

The large influence that electronics and software have on the vehicle's character means that errors must be avoided. This is necessary to maintain confidence in a brand and its electronics functions and to avoid recall costs and expensive reengineering. As safety-related functions are increasingly dependent on software, avoiding errors is also a safety issue. Specification and configuration faults are common and cause severe failures, even compared to implementation faults and random component faults. Means to improve the development methodology in a way that can reduce these errors are thus important.

9.2.5 Concurrent Engineering

The reduction of development lead times means that a traditional waterfall development is not possible. Instead, people are concurrently and iteratively working on development aspects that belong to different phases of the ideal product development cycle. This mode of work requires consistent and up-to-date information.

9.2.6 Reuse and Product Line Architectures

Reuse of functions and subsystems is necessary to capitalize on the development cost and scale economy that can be achieved. To understand what an existing function does, to know its interface and resource consumption, requires effective means of specifying functions. It is also important to be able to distinguish between implementation-specific parts, and the generic parts that can be reused.

Component-based design is an important approach, which is widely used. Full use of component-based design requires adequate specifications of the components, interfaces, and the required infrastructure.

9.2.7 Analysis and Synthesis

Reducing lead time with sustained quality can only be achieved by using state-of-the-art analysis and synthesis techniques. The engineering information related to embedded system development must therefore include the input data required for the various analysis and synthesis methods that apply, as well as the output data provided by these methods.

9.2.8 Prototyping

Early assessment of systems and products under development is important to validate that the right system is being developed. Virtual prototyping, rapid control prototyping, hardware-in-the-loop (HiL), and software-in-the-loop are all state-of-the-art techniques that make it possible to find and eliminate misunderstandings and mistakes in specifications as well as errors during implementations. These techniques rely on precise (although evolving) information about the systems and their interfaces.

These were some key features of the engineering information challenges that apply, in particular, in a future scenario with advanced methodology and tools. The next section will discuss current approaches.

9.3 State of Practice

The last decade in embedded automotive software development was characterized by the introduction of a standardized operating system as a first step to create an electronic control unit (ECU) software architecture, the introduction of the V-cycle development approach as well as the introduction of model-based design in conjunction with rapid prototyping. This section will explain how this is presently done.

9.3.1 Model-Based Design

The development of embedded automotive control software is characterized by several development steps that can be summarized by using the V-model [1]. One starts with the analysis and design of the logical system architecture, that is, defines the control functions, proceeds with defining the technical architecture, which is a set of networked ECUs, and then proceeds with software implementation on an ECU. The software modules will be integrated and tested, then the ECU is integrated in the vehicle network and last but not least, the system running the implemented functions is fine-tuned by means of calibration. However, this is not a top-down process, but requires early feedback by means of simulation and rapid prototyping.

9.3.1.1 Control Algorithm Development

At first, control algorithms are developed. This is mainly a control-engineering task. It starts by the dynamic analysis of the system to be controlled, that is, the plant. A plant model is a model of the vehicle (including the sensors and actuators), its environment (e.g., the road conditions), and the driver. Typically, only subsystems of the vehicle are considered in special scenarios like the engine with the power train and the driver, or the chassis with the road conditions. These models can be either analytical, such as an analytically solved differential equation, or a simulation model, that is, a differential equation to be solved numerically. In practice, a model is often a mixture of both.

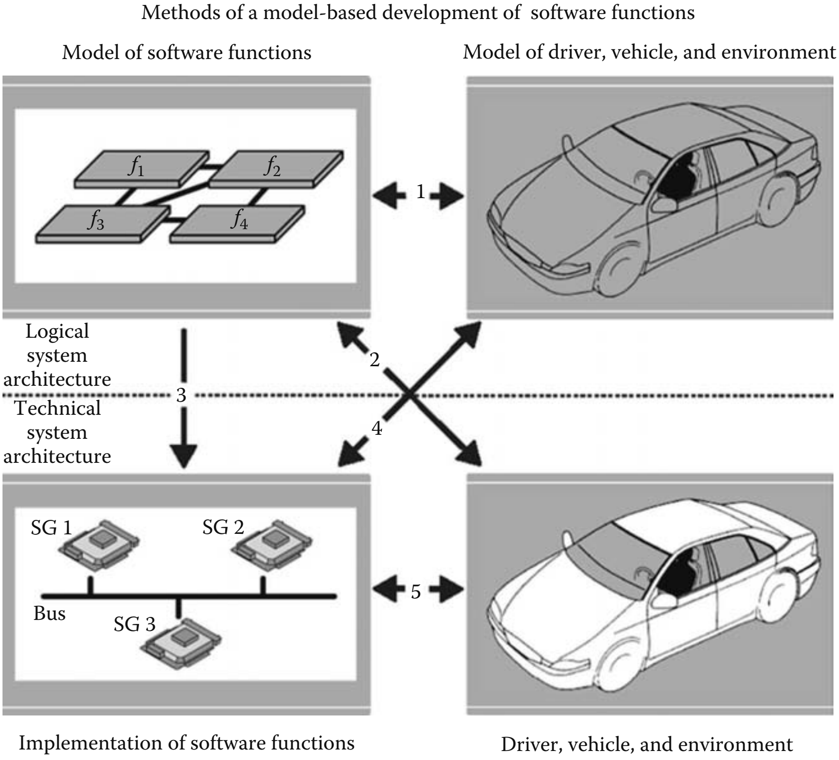
Then, according to some quality criteria, the control law is applied. Control laws compensate the dynamics of a plant. There are many rules to find good control laws. Automotive control algorithms very often combine closed-loop control laws with open-loop control strategies. The latter are often automaton or switching constructs. This means that control algorithms are hybrid systems from a system-theory point of view. Typically, the control law consists of set-point-generating function with controlling and monitoring functions, all realized by the software. The first step is to design a control algorithm for a vehicle subsystem, which is represented as a simulation model. Both the control algorithm and the plant model are running on a computer. The plant is typically realized as a quasicontinuous time model while the control algorithm is modeled in discrete time. The value range of both models is continuous, that is, the state variables and parameters of the control algorithm and the plant are realized as floating point variables in the simulation code. This model is depicted in the upper part of Figure 9.1. The logical system architecture represents the control algorithm and the model of driver, vehicle, and environment of the plant. The arrow labeled 1 represents the control algorithm design step.

9.3.1.2 Rapid Prototyping

Unfortunately, the employed plant models are typically not detailed enough to serve as a unique reference throughout the design process. Therefore, the control algorithm has to be checked in a real vehicle. This is the first time the control algorithm will run in real time. Therefore, the executable parts of the time-discrete simulation model have to be chosen carefully. The executable parts have to be mapped to operating system tasks while dedicated software modules for hardware access have to be attached to the control algorithm.

This step is shown in Figure 9.1 in linking the logical system architecture to the real vehicle, which is driven by a driver in a real environment, represented by the arrow labeled 2. There are many ways to realize this step. First of all, one can use a dedicated rapid prototyping system with dedicated input/output (I/O) boards to interface with the vehicle. The rapid development systems (RP-system) consist of a powerful processor board and I/O boards as depicted. The boards are connected via an internal bus system. Compared to a series-production ECU, these processor boards are generally more powerful, have floating point arithmetic units, and provide more read only memory (ROM) and random access memory (RAM). Interfacing with sensors and actuators via bus-connected boards provides flexibility in different use-cases. In brief, priority is on rapid prototyping of control algorithms and not on production cost of ECUs.

The interfacing needs of the rapid development systems often result in dedicated electrics on the boards. This limits flexibility, and an alternative is therefore to interface to sensors and actuators using a conventional ECU with its microcontroller peripherals and ECU electronics. A positive side effect is that the software components of the I/O-hardware abstraction layer can be reused for series production later on.


Methods of a model-based development of software functions
Model of software functions
Model of driver, vehicle, and environment
f_1
f_2
f_3
f_4
1
2
3
4
5
Logical system architecture
Technical system architecture
SG 1
SG 2
SG 3
Bus
Implementation of software functions
Driver, vehicle, and environment

1. Modeling and simulation of software functions as well as of the vehicle, the driver, and the environment.
2. Rapid prototyping of software functions in the real vehicle.
3. Design and implementation of software functions.
4. Integration and test of software functions with lab vehicles and test benches.
5. Test and calibration of software functions in the vehicle.

FIGURE 9.1 Model-based development of a software function.

Figure 9.2 shows that the control and monitoring functions run on a bypass system, which is connected via sensors and actuators to the vehicle.

For rapid prototyping in bypass configuration as shown in Figure 9.2, the ECUs μC-peripherals are used to drive the sensors and actuators. This means, the control algorithm still runs in the rapid development hardware whereas the I/O drivers are running on the series production ECU. The signals W, R, and U are digital values representing the set point, the sampled reaction of the plant, and the digital actuator signal. The actuator signal is transformed to an electrical or mechanical signal Y driving the vehicle in the state prescribed by the driver's wish W^*. W is the sampled digital signal. The actual state of the vehicle in terms of mechanical or electrical signals X is sampled and fed to the control algorithm as digital signal R. Furthermore, there are noise signals Z like the road conditions that are not directly taken into account by the control algorithm as measured input signal, but also influence the behavior of

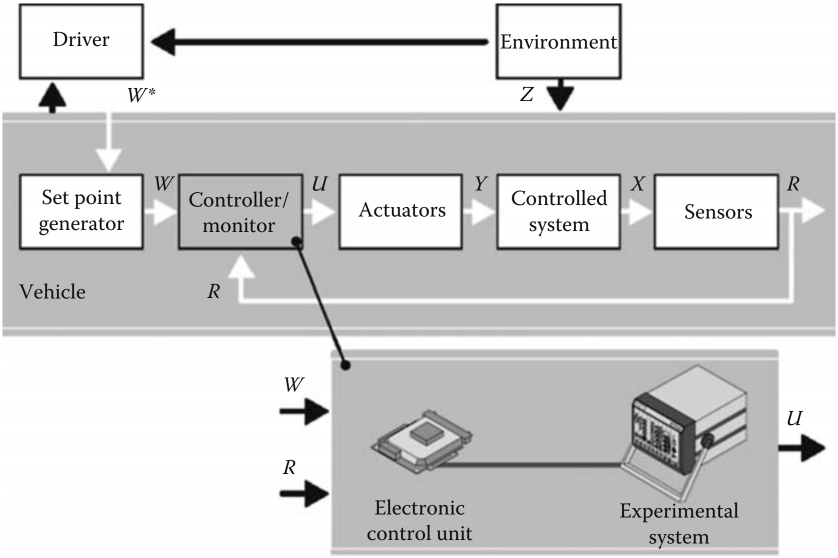


FIGURE 9.2 A typical rapid prototyping system.

the vehicle. Provided no other vehicle signals are used directly, the RP-system uses only a dedicated communication board in addition to the processor board. The sensor values *R*, the set point values *W*, and actuator values *U* are transmitted over the high-speed link. In most cases, the ECU hardware is modified with dedicated facilities to accommodate the high-speed communication link.

9.3.1.3 Implementation and ECU Integration of Control Algorithms

After the rapid prototyping step, the control algorithm is proven valid for use in the vehicle. The code that was generated for rapid prototyping systems relied on the special features of the processing board, such as RAM resources and the floating point unit. To make the control algorithm executable under limited memory and computational resources, the model of the control algorithm has to be reengineered. For example, computation formulas are transformed from floating point to fixed point control algorithms, and efficiency scalability, modularity, and other concerns are addressed. The adapted design can be automatically transformed to production code in a code-generation step.

9.3.1.4 Testing the Technical System Architecture in the Laboratory

The result of the implementation and integration phase is the technical system architecture, that is, networked ECUs. These ECUs are tested against plant models in real time. The plant models themselves are augmented by models of the sensors and actuators and dedicated boards being able to simulate the electrical signals as they are expected by the ECU electronics. These kinds of systems are called hardware-in-the-loop systems and consist of processing and I/O boards. The plant model is initialized with a lot of different values simulating typical driving maneuvers. Then, the driving maneuver is simulated on the HiL as providing ECU sensor data as output and accepting ECU actuator data as input. This way it can be checked whether the ECU integration was successful. HiL testing is represented by the arrow labeled 4 in Figure 9.1.

9.3.1.5 Testing and Honing of the Technical System Architecture in the Vehicle

As written above, there are many use-cases where plant models are not detailed enough to represent the vehicle's dynamics. Though a lot of calibration activities can nowadays be done by means of HiL systems, final honing of a vehicle's control algorithm still needs to be done with the production software in a production ECU in a real vehicle. This requires that the technical system architecture be built into a vehicle and tests be done on proving ground. This kind of fine-tuning only concerns the parameter settings of the control algorithm, not its structure.

9.3.2 Tools

To illustrate tools used for automotive software development, ASCET [2] will be described. ASCET uses an object-based real-time paradigm to construct embedded automotive control software. The main building blocks are classes for the functional design and modules for the real-time design. To constitute an ECU, modules are aggregated in projects. A class aggregates methods having arguments and return values. Classes provide an internal state by means of variables and can aggregate instances of other classes. Inheritance is not supported. There are two flavors of classes, "simple" classes and "finite state machine" classes. For embedded real-time modeling, there are modules that provide messages as means to transfer data between modules. Modules use instances of classes. Modules provide so-called processes that read and write messages and call methods of a class. The processes are realized in C-Code as subroutines having no formal arguments. This allows efficient implementation of the interprocess communication but restricts the number of module instances per ECU to one.

Figure 9.3 shows a brake-slip-control algorithm as ASCET module. It employs for each wheel an instance of the class-slip-control. The wheel speeds as well as the vehicle speed are provided as receive message of a module. The control pressure for each wheel is provided as send message. Messages realize a thread-safe communication mechanism, which means that every message will be guarded by embedded real-time interprocess communication means. Projects collect all modules that run on an ECU. The throttle-control algorithm shown in Figure 9.4 shows eight modules. These modules provide 13 processes. Some of these processes are only used for initialization. The execution scheme of all modules on that ECU is provided by allocating the processes to tasks as shown on the left-hand side of Figure 9.5. The messages of the modules are connected by name-matching on the ECU global project level. The resulting data flow is shown in the connections of Figure 9.4 and listed as a table on the right-hand side in Figure 9.5.

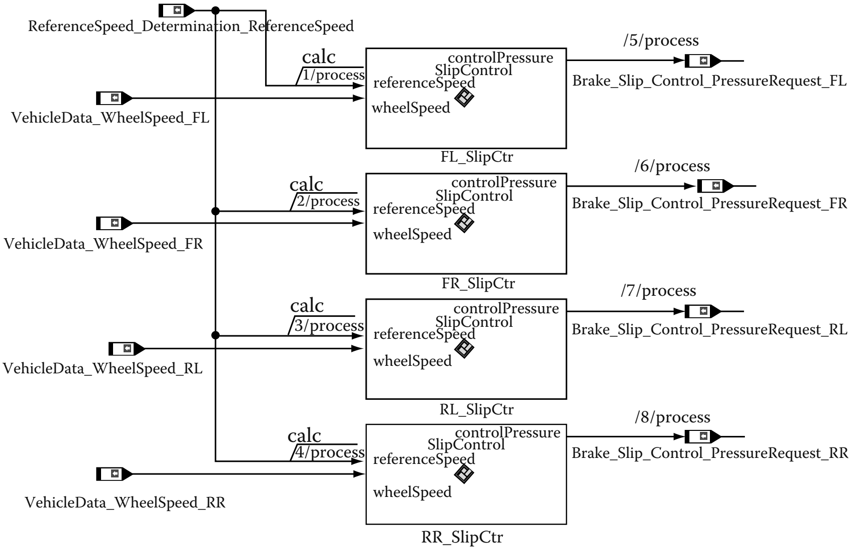

ReferenceSpeed_Determination_ReferenceSpeed
VehicleData_WheelSpeed_FL
VehicleData_WheelSpeed_FR
VehicleData_WheelSpeed_RL
VehicleData_WheelSpeed_RR
calc
/1/process
/2/process
/3/process
/4/process
controlPressure
SlipControl
referenceSpeed
wheelSpeed
FL_SlipCtr
FR_SlipCtr
RL_SlipCtr
RR_SlipCtr
/5/process
/6/process
/7/process
/8/process
Brake_Slip_Control_PressureRequest_FL
Brake_Slip_Control_PressureRequest_FR
Brake_Slip_Control_PressureRequest_RL
Brake_Slip_Control_PressureRequest_RR

FIGURE 9.3 Brake-slip-control algorithm in ASCET.

The execution paradigm of ASCET project is: A task calls a process. The process reads from receive messages and calls the methods of the class instances by passing message data to arguments. Then, the processes call other methods of the instances receiving return values that will then be written to send messages. Then the next process in the task is called. Data integrity is ensured by copying mechanisms of ASCET messages.

ASCET models are transferred to executable code by applying code generation to projects that will generate the code for the modules and classes. For production code generation, fixed point arithmetic is applied as shown in Figure 9.6. For example, the wheelslip determination of a very simple antilock braking system (ABS)[*] control algorithm as shown in Figure 9.7 where the wheelslip is given by the difference of the wheel velocity and the vehicle velocity (represented by the reference speed) and normalized by the limited vehicle velocity (represented by the reference value).

The resulting code is generated in a straightforward manner:

```
self-> WheelSlip-> val = (referenceSpeed - wheelSpeed)/self->
  ReferenceValue-> val;
```

This changes completely when the floating point arithmetic is abandoned and the fixed-point arithmetic is introduced. This means not only to use integer values instead

[*] If a wheel locks under braking, the brake opens for some microseconds to let the wheel regain velocity with respect to the vehicle's actual speed. Only running wheels can carry lateral forces that allows vechicles to steer and, for example, to avoid obstacles on the road.

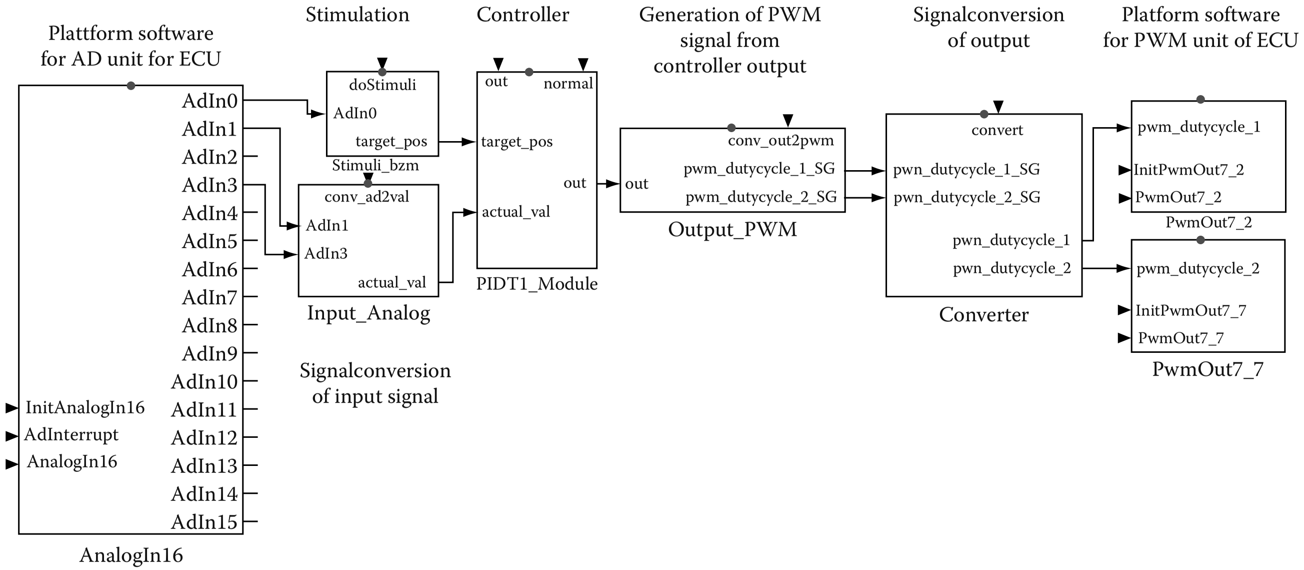
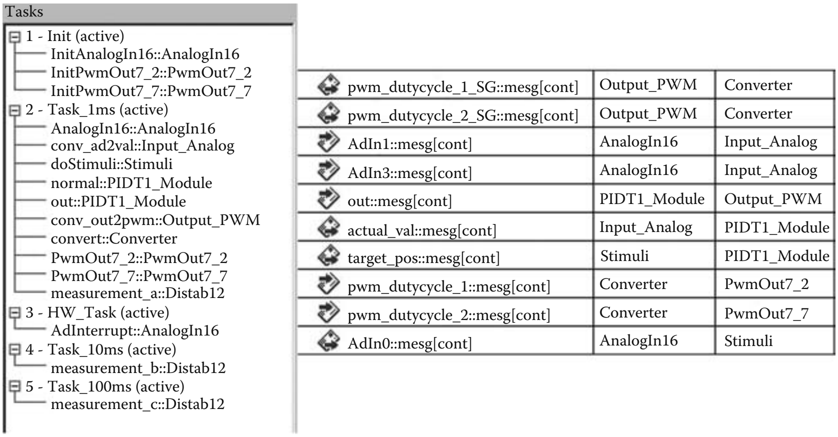


FIGURE 9.4 Simple model of a throttle controller in ASCET.

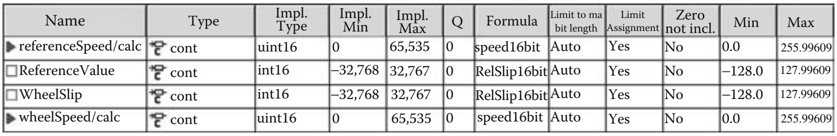


Tasks

- 1 - Init (active)
  - InitAnalogIn16::AnalogIn16
  - InitPwmOut7_2::PwmOut7_2
  - InitPwmOut7_7::PwmOut7_7
- 2 - Task_1ms (active)
  - AnalogIn16::AnalogIn16
  - conv_ad2val::Input_Analog
  - doStimuli::Stimuli
  - normal::PIDT1_Module
  - out::PIDT1_Module
  - conv_out2pwm::Output_PWM
  - convert::Converter
  - PwmOut7_2::PwmOut7_2
  - PwmOut7_7::PwmOut7_7
  - measurement_a::Distab12
- 3 - HW_Task (active)
  - AdInterrupt::AnalogIn16
- 4 - Task_10ms (active)
  - measurement_b::Distab12
- 5 - Task_100ms (active)
  - measurement_c::Distab12

pwm_dutycycle_1_SG::mesg[cont]	Output_PWM	Converter
pwm_dutycycle_2_SG::mesg[cont]	Output_PWM	Converter
AdIn1::mesg[cont]	AnalogIn16	Input_Analog
AdIn3::mesg[cont]	AnalogIn16	Input_Analog
out::mesg[cont]	PIDT1_Module	Output_PWM
actual_val::mesg[cont]	Input_Analog	PIDT1_Module
target_pos::mesg[cont]	Stimuli	PIDT1_Module
pwm_dutycycle_1::mesg[cont]	Converter	PwmOut7_2
pwm_dutycycle_2::mesg[cont]	Converter	PwmOut7_7
AdIn0::mesg[cont]	AnalogIn16	Stimuli

FIGURE 9.5 Task process schedule (left) and message flow in the throttle controller (right).



Name	Type	Impl. Type	Impl. Min	Impl. Max	Q	Formula	Limit to ma bit length	Limit Assignment	Zero not incl.	Min	Max
▶ referenceSpeed/calc	cont	uint16	0	65,535	0	speed16bit	Auto	Yes	No	0.0	255.99609
□ ReferenceValue	cont	int16	–32,768	32,767	0	RelSlip16bit	Auto	Yes	No	–128.0	127.99609
□ WheelSlip	cont	int16	–32,768	32,767	0	RelSlip16bit	Auto	Yes	No	–128.0	127.99609
▶ wheelSpeed/calc	cont	uint16	0	65,535	0	speed16bit	Auto	Yes	No	0.0	255.99609

FIGURE 9.6 Annotation for 16 bit implementation.

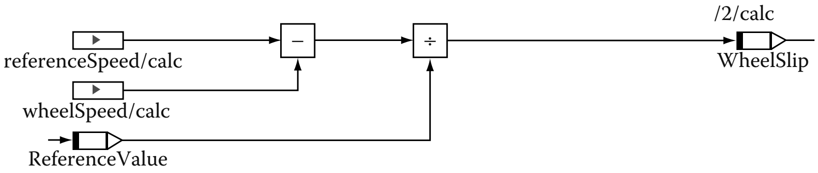


FIGURE 9.7 Model extract for wheelslip determination.

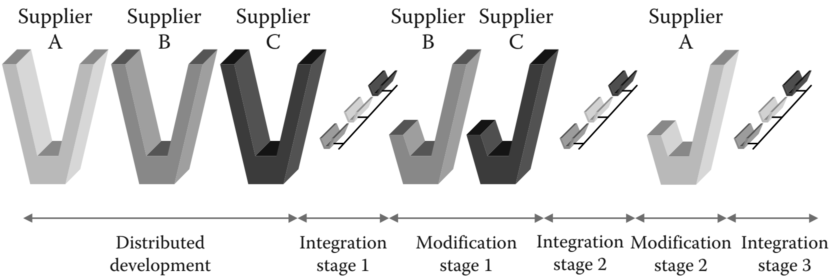


FIGURE 9.8 Integration problems in ECU networks.

of floats, but also to imply a physical meaning. For example, the reference speed ranges from 0 to 256 km/h and is represented by bit patterns from 0 to 65,535, which associates to 1/8 km/h per bit. Of course, this quantization has to be reflected by the generated C-Code, where additional operations are introduced.

```
_WheelSlip = (sint16)((referenceSpeed - wheelSpeed << 8)/
  (sint32)_ReferenceValue);
```

ASCET can be used as a behavioral modeling tool in AUTomotive Open System ARchitecture (AUTOSAR) [3] (Chapter 2) or other ADLs. From the integration point of view, clustering plays an important role. In AUTOSAR systems, for example, processes are grouped to runnable entities while several modules establish the internal behavior of atomic software components. The feasibility of this approach has been shown on integrating a cruise-control function on an AUTOSAR ECU [4], thus leveraging the advantages of model-based design and ADLs.

9.3.3 Problems beyond Model-Based Design

In the past, model-based design was mainly used for the design of single ECUs and led to significant quality and productivity improvements. However, the interoperability description of all functions is still based on a communication matrix of all controller area network (CAN) messages, which is given in parts to the appropriate supplier. Then, every supplier involved will start their own development cycle according to the V-model. This situation is depicted on the left-hand side of Figure 9.8. The first integration attempt (Integration Stage 1) with the final ECUs at the vehicle manufacturer's premises usually leads to a failure. As a rule, some suppliers have to redesign their system only to see it failing again at the next integration (Integration Stage 2), for example, due to problems with another ECU.

Analyzing the reasons for failure in the integration phase exhibits the following problems: timing problems, interface problems, and communication problems between the ECUs. All problems are interrelated and need to be solved at the appropriate level of abstraction. The introduction of a configurable middleware layer in the ECU software architecture as well as an ADL to provide the configuration information is currently seen as the solution to the integration problems [5]. Furthermore, the ADL has to integrate the model-based design approach to leverage design advantages already existing in the current ECU-centric design approach.

9.4 ADL as a Solution

To progress from current best practices into a situation where the engineering information is managed in a more effective way, an ADL is useful. Architecture is defined by the IEEE as "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." An ADL is in its widest sense an approach to documenting the architecture in a semantically precise way. For the domain of automotive electronics and software, an ADL should address the challenges of engineering information management identified above. We will continue by discussing how this can be done.

9.4.1 General Aspects on an Automotive ADL

An ADL that constitutes an agreed approach to documenting automotive software and electronic architectures serves several purposes. The language provides an ontology that helps reasoning about the systems among the stakeholders. A supplier and OEM, or engineers in different departments will thereby be able to understand each other's specifications immediately. The learning curve of new approaches and notations will thereby be less steep.

With semantically and syntactically precise description along with agreed exchange formats, it will be possible to exchange descriptions between different tools. To reach this goal, standardization is important, irrespective of whether it is a de facto standard or a formal standard.

To capture the different aspects of an automotive component, system, or entire vehicle, several aspects need to be modeled. This can be achieved by including many aspects in the same model, or by providing separate models depending on concerns. Examples of concerns are hardware versus software, system versus environment, and abstract versus concrete descriptions. In the former case, where an integrated model is used, the complexity of those models makes it necessary to organize them in a way that provides separation of concerns. If different models are used, the definition of views and viewpoint as defined in IEEE 1471 [6] and systems modeling language (SysML) [7] is useful. The draw-back in using separate models is that the traceability between aspects is lost. If possible, the ADL should therefore address several aspects in one integrated model.

We will continue with a discussion of the modeling support expected of an automotive ADL.

9.4.2 What Needs to Be Modeled

The engineering information that an automotive ADL should cover include structure, behavior, requirements, and variability as well as support for verification and validation and synthesis. These categories of information are ideally covered entirely by an ADL, to allow for information integration and traceability between parts of the information. Alternatively, a well-defined set of description means can be defined where, for example, the ADL covers structure and behavior and other aspects are handled by external tools and notations. Below, we will describe each of these areas.

9.4.2.1 Structure

By structure we mean the logical or physical organization of the modeled system. The structure is thus different depending on the aspect of the system that is considered. It may be the functional decomposition with functions, ports, and connections or the software architecture with its software components and interfaces. For hardware it means the physical components and connectors. In each case, the structure is the backbone of the system description, and modeling entities for other system information should relate to the structural entities. For example, behavioral models, requirements, and validation/verification models can be linked to the relevant functional, software, or hardware component.

9.4.2.2 Behavior

Modeling behavior is in one sense even more important than structure, since it defines what the component or entire system should do. Behavioral models also have the role to capture the external environment in order to allow validation and verification in a realistic context.

The nature of behavioral models differ depending on these two purposes. Environment models should describe the external world, or plant, within the vehicle and outside the vehicle. The detail level should be sufficient to support the analysis at hand, while maintaining sufficient simulation or analysis performance. Parts of the environment model may therefore occur in several variants to match different needs.

Behavioral models for the software-based electric/electronic system are of at least three different kinds: (1) abstract descriptions of behavior for early or implementation-independent simulation and analysis; (2) concrete descriptions of behavior for implementation-oriented simulation, analysis, and code generation; and (3) actual code for simulation, analysis, and compilation to target system.

The decomposition of the behavioral definitions should ideally match the decomposition of the structure, that is, each structural component should have a corresponding behavioral definition. In case this is not possible, it should be possible to identify which set of structural components correspond to which behavioral definition(s).

It is also highly important that behavioral definitions of different components be composed to a behavioral model that corresponds to the complete system or a subset of it.

9.4.2.3 Requirements

System requirements represent an important category of engineering information, and should thus be supported by the ADL. In addition, requirements are related to other requirements and to the system entities they constrain. Including requirements in an ADL makes it possible to capture the traceability between requirements and between components and their requirements. With this modeling support, it is possible to identify which requirements were the underlying reasons for a particular refined requirement, or to see what requirements constrain a particular component.

Requirements may be textual or model based, informal or formal. By model based we mean that the requirement is expressed using modeling entities in the ADL, such as expressing a timing requirement by referring to events on ports, or expressing an allocation requirement by referring to the target hardware and the allocated software.

9.4.2.4 Variability

Variability is a highly important aspect on automotive software and systems development. Variability needs to be taken into account from the beginning of development to allow reuse, define appropriate product lines, and ensure correctness of configurations. Another role of variability is to define different configurations during development, for example, for a particular review or analysis.

System models need to capture the needs and requirements that motivate differentiation between products. It is also necessary to define the traceability between variability in components and the variability that is visible externally. Finally, managing the component variability is important, that is, defining the variants and defining the rules for when the respective variant is chosen. With a clear representation of variability, the implementation in terms of parameterization, hardware, and software component inclusion can be managed appropriately. Variability is covered in more depth in Chapter 7.

9.4.2.5 Verification and Validation

Verification and validation of vehicle systems are done in several development phases and involve different parts of the vehicle. The purpose is to validate for the marketing department whether the vehicle's functions are satisfactory or to verify whether the system requirements have been satisfied in specifications and implementations. The techniques that may be used involve inspection, simulation, prototyping, testing, and numerical and formal analyses.

Important simulation techniques include HiL (complete vehicle system is validated with an interface to remaining vehicle), software-in-the-loop (software of vehicle system is validated with an interface to remaining vehicle including hardware and platform software of local control unit), and rapid control prototyping (functional representation of vehicle system with interface to sensors and actuators of the actual vehicle, allowing validation in real vehicle).

Many of these techniques require models of the system under development as well as its surrounding systems and the vehicle's mechanical parts. Other engineering information that can be supported by an ADL includes definitions of the verification/validation setup, input stimuli, and representation of verification outcome and verdict. A particularly useful aspect of ADL support is that the verification setup, inputs, and outputs can be rigorously documented.

Examples of analyses that benefit from a model-based representation of the system are safety analyses such as fault tree analysis and failure modes and effects analysis; timing analysis such as bus schedulability analysis and CPU response time analysis; and formal verification for verifying formally expressed properties and system behavior.

9.4.2.6 Synthesis

The development process from user needs over requirements specifications to final implementations can be seen as a continuous refinement of engineering information. In some of these steps, automatic synthesis is possible, most notably code generation out of more abstract behavioral definitions. Other examples of synthesis include automatic software-to-hardware allocation or automatic generation of tests. These synthesis steps require adequate representation of the input information.

For code generation it means behavioral representations that are ideally part of the ADL, or means to identify behavioral representations in external notations. Allocation requires representation of the software and its safety requirements, resource requirements for computation, memory and communication, various requirements that constrain allocation, end-to-end response time analysis, etc. For hardware the same characterization is needed in terms of what it provides. Test generation as a final example, requires representation of the requirements and properties the test shall verify and representation of the behavior and interfaces of the system under test. In each of these cases, the level of detail in the representation affects the accuracy of the results. For example, a preliminary and abstract signal-to-link allocation may have to be changed when the final frame set and bus capacity are compared.

9.5 Existing ADL Approaches

This section describes five approaches that are relevant for modeling of automotive electronic systems.

9.5.1 Forsoft Automotive

The Forsoft Automotive project [8] integrates requirements management, software-component design, and behavioral modeling by means of the automotive modeling language (AML). The AML is a metamodel and supports five abstraction levels:

1. Signals
2. Functions
3. Logical architecture
4. Technical architecture
5. Implementation

Signals represent interfaces that are referenced by functions. The composition of functions represent the logical architecture realizing the control algorithm while the technical architecture comprises the whole physical network of ECUs in addition to the basic software modules running on that ECU.

AML representations of the first three abstraction levels are given by the unified modeling language (UML) modeling tool, UML suite, and the graphical modeling and code-generation tool, ASCET. The AML description is based on interconnected functions. A function is composed of ports that are typed by interfaces. Interfaces carry signals. Functions can be grouped hierarchically. Functions can have variants. A variant of a function implements only subsets of the function with respect to ports. Interfaces can also have variants where the variants are always subsets. This concept is shown in more detail in Figure 9.9. At the top, there is the function window lifting with its signal interfaces interrupt, window–control, and comfort control. The driver door function provides interfaces for window and comfort control whereas the variants codriver door and passenger door just employ the window–control interface. All variants employ the interrupt interfaces.

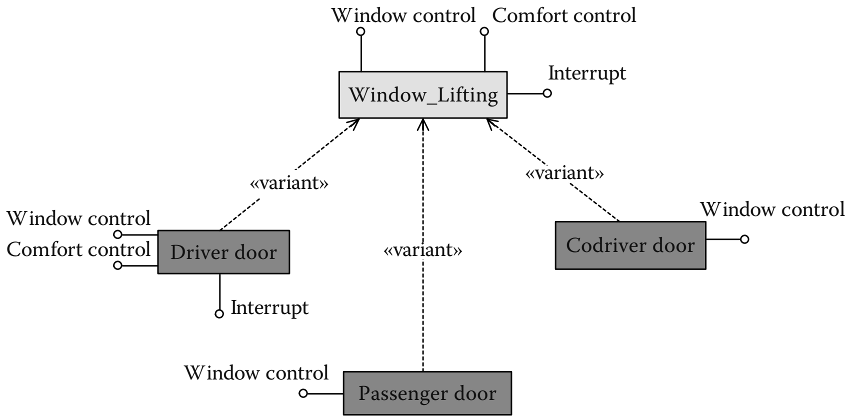


FIGURE 9.9 Variant concept based on subsets.

The logical architecture is represented by a variant of the hierarchical function. The technical architecture is given by networked ECUs including all basic software modules for communication in addition to the leaf functions of the logical architecture. Of course, all delegation and propagation connectors are resolved in the mapping step. Implementation means code generation of the leaf functions as well as manual coding or configuration of basic software modules realizing the interface between the ECU hardware and the leaf functions of the control algorithm.

9.5.2 SysML

SysML [7] is the Object Management Group (OMG) systems modeling language (Figure 9.10). It was initiated in 2003 as a response to the OMG Systems Engineering

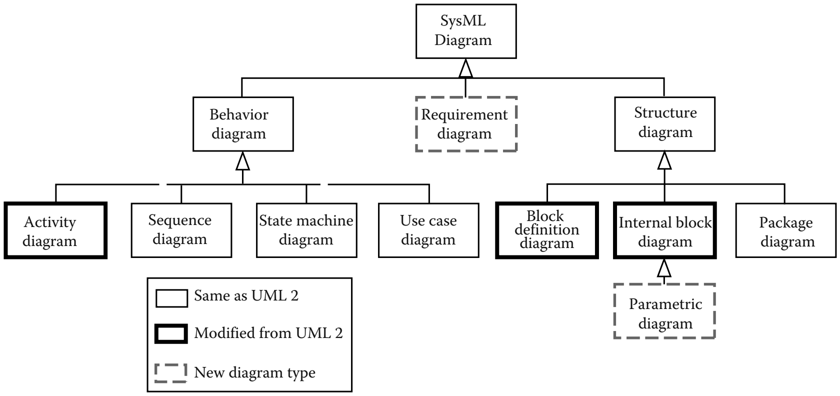


FIGURE 9.10 The SysML diagram taxonomy. (From Object Management Group, Systems Modeling Language (SysML) Specification, ptc/06-05-04, 2006. With permission.)

Request for Proposal, with early participation from tool vendors, defense, aerospace, and agriculture industry. It is a UML profile extending UML2 version 2.1, and also complies to data interchange according to STEP (ISO 10303-233).

SysML extends UML2 with new modeling concepts and diagrams for systems engineering. It also identifies the parts of UML2 that are relevant for systems engineering modeling. The new concepts concerns requirements, structural modeling, and behavioral constructs. New diagrams include requirement diagram and parametric diagram, and adjustments of activity, class, and composite structure diagrams. Tabular representations of requirements or allocations, for example, are also included as an alternative notation.

SysML is a generic approach, which can and should be specialized for the respective domain where it is used. This is possible, since the language is a UML2 profile that can be further extended.

9.5.3 Architecture and Analysis Description Language

AADL is the SAE architecture and analysis description language [9]. It is based on the MetaH, a language for the definition of software and hardware components and their allocation. The origin of MetaH and the AADL is defense and aerospace domains, but the target is currently embedded systems in general. It is an SAE standard since 2004.

The focus of AADL is on task structure and interaction topology, although generalization to more abstract entities is possible. It supports the definition of mode handling, error handling, interprocess communication, etc. As such, it acts as a specification of the embedded software, which can be used for autogeneration of an application framework where the actual code can be integrated smoothly. The language contains various analysis support, for example, for safety and timing. Further, a behavioral annex is proposed, to allow a common behavioral semantics for AADL descriptions.

Currently, the AADL is the subject of several research efforts where tools and analysis extensions are investigated, for example, the TOPCASED project in France [10] or work at the SEI in the United States [9].

9.5.4 Modeling and Analysis of Real-Time and Embedded Systems

The UML profile for modeling and analysis of real-time and embedded systems (MARTE) [11,12], is an approach to modeling real-time and embedded systems in UML2, and to support analysis of relevant properties (Figure 9.11). Both hardware and software aspects are supported. MARTE is currently an OMG request for proposal, with a major submission from the proMarte consortium [13]. It is meant to replace the current profile for schedulability, performance, and time [14].

The Marte profile is organized in three major packages: time and concurrent resource, runtime environment (RTE) modeling, and RTE analysis. Time and concurrent resource is a kind of infrastructure for MARTE modeling and analysis. RTE modeling extends UML2 with constructs for modeling-embedded software,

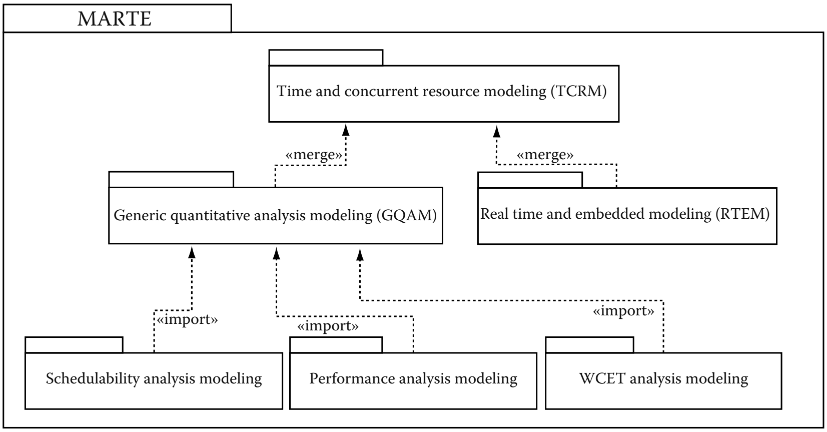

MARTE
Time and concurrent resource modeling (TCRM)
«merge»
«merge»
Generic quantitative analysis modeling (GQAM)
Real time and embedded modeling (RTEM)
«import»
«import»
«import»
Schedulability analysis modeling
Performance analysis modeling
WCET analysis modeling

FIGURE 9.11 Modeling and analysis of real-time and embedded systems, profile architecture.

hardware, and allocation. The RTE analysis extends UML2 with concepts for schedulability and performance analysis.

The Marte profile is not an ADL, but together with the UML2 constructs, much of the modeling needs of an ADL is covered.

9.5.5 AUTOSAR Modeling

As written above, ECU software-architectures employing a middleware and using an ADL for configuration are seen as means to overcome integration problems in networked systems. AUTOSAR proposes an RTE as middleware and a virtual functional bus (VFB) as ADL. On the VFB, all control algorithm-related interface problems are resolved. The VFB is organized in hierarchically interconnected software components. In a mapping and configuration step, the VFB is transformed to networked ECUs running parts of the control algorithm.

The main concepts of AUTOSAR are

- VFB view of interconnected application software components
- Software component description of interfaces, internal behavior, and runnables
- ECU network description
- System constraint description with premapped signals
- Mapping description of software-components to ECUs, connectors to frames, and runnables (the concurrency elements of software components) to tasks
- Layered ECU software architecture with configurable basic software modules
- Configuration description of the basic software modules

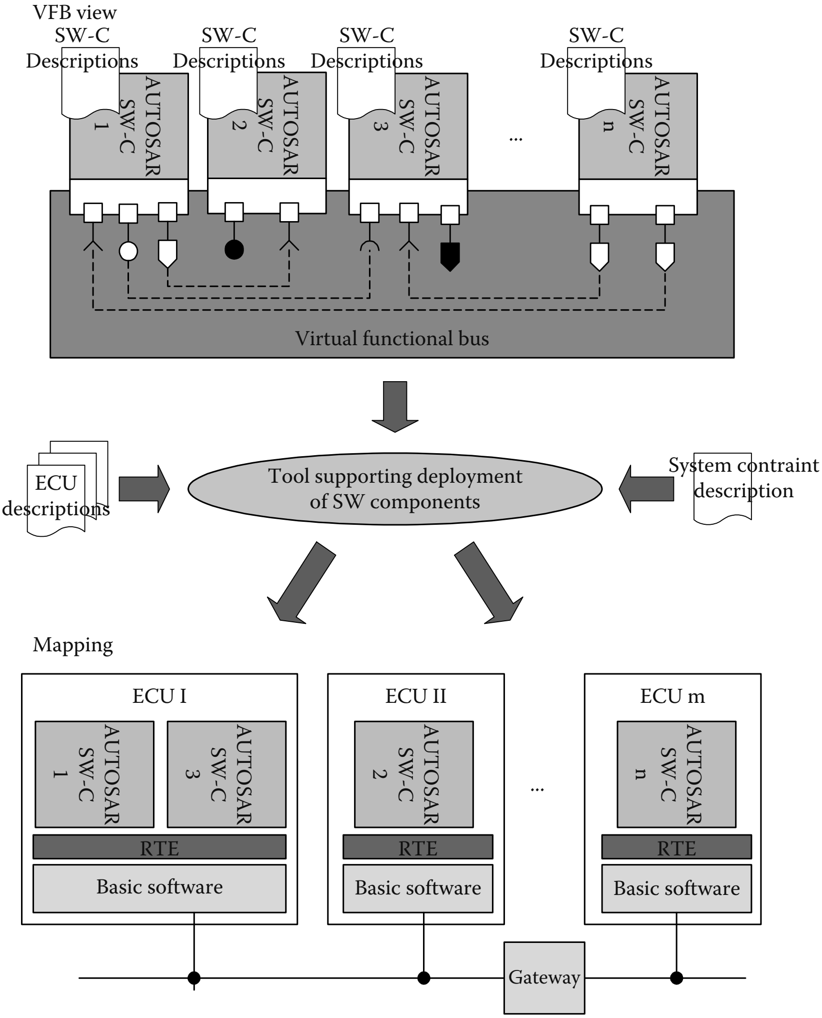


FIGURE 9.12 The AUTOSAR approach.

The interaction of these concepts is shown in Figure 9.12. At the top, there are the application software components with their ports. The ports are typed by interfaces and can be either of sender/receiver type (arrow shape) or of client/server type (UML lollipop notation). Additionally, there are so-called service ports that interface to standardized ECU services like nonvolatile RAM (NVRAM) management or diagnostics. Ports of the application software components are connected by connectors. These connected application software components establish the VFB representation of the system and contain the control algorithms. It is virtual because no assumptions are made on the underlying E/E-architecture. It can be designed completely independently and allows a relocation of software-components from one ECU to another in different vehicle types. The E/E-architecture is reflected by the ECU resource description and the system-constraint description. Both are used to describe the type and characteristics of the employed ECUs, networks, and gateways. It is shown in the middle of Figure 9.12. The system constraint description might also contain a mapping description of system signals to frames. It therefore constitutes a partial communication matrix. According to a chosen mapping criterion, application software components are mapped to ECUs. All connectors, connecting software components mapped to different ECUs, are subject to signal mapping. This means that the data elements of the port's interface are allocated to frames on the bus system. After all of these "remote" connectors have been mapped, the communication matrix is established and the communication stack of all ECUs can be configured. All connectors connecting software components allocated to the same ECU will be realized by the RTE. The RTE is configured according to the software component and mapping description and provides real-time embedded interprocess-communication means. These means are hidden by macros so that a runnable of a software component can read data from the RTE or write data to the RTE. To provide an optimal implementation of the RTE, it is configured taking into account the software-to-task allocation and the task schedule. In highly preemptive systems the resource consumption of an RTE might be higher than in a cooperative system. The resulting system is shown in the lower part of Figure 9.12. Here all software components have been mapped to ECUs, the RTE, the COM stack, and all other basic software modules are configured.

9.5.6 EAST-ADL

The EAST-ADL was developed in the ITEA project EAST-EEA [15]. The EAST-EEA project was about reducing the hardware dependency of automotive software, allowing more flexibility regarding the allocation of software. It was recognized that an ADL was needed in order to manage the specifications of software and hardware. The EAST-ADL has subsequently been applied in e.g., the EASIs project [16] and refined in the ATESST project [17] to EAST-ADL2.

The scope of the EAST-ADL is the engineering information required for developing automotive software: the functions and software itself, requirements, hardware, and environment. Figure 9.13 shows the abstraction levels that are used to organize the system information in the EAST-ADL2.

9.5.6.1 System Model Organization

Vehicle Level

This abstraction level represents the user's perception of the vehicle. The model contains vehicles and the features that it offers. Requirements, expressed on an abstract, implementation-independent form may be assigned. It is possible to configure the vehicle, that is, define which features are included in a particular vehicle.

Analysis Level

This level contains models with an abstract description of the functionality of the EE architecture. Focus is on what the systems should do, rather than how, although some choices are made. For example, a preliminary functional decomposition is made

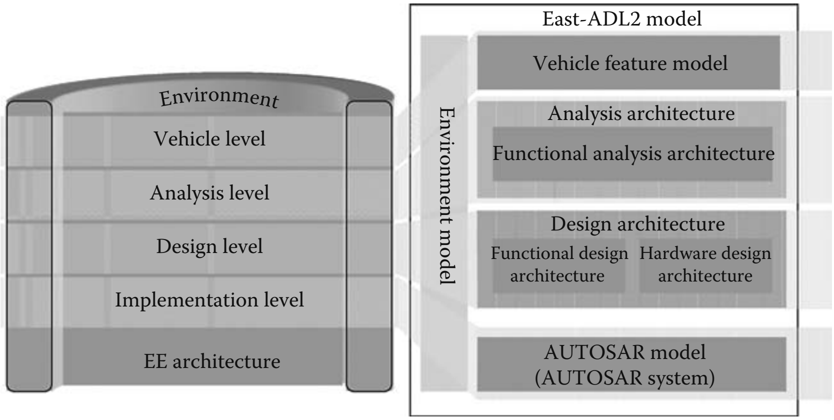


FIGURE 9.13 The EAST-ADL2 abstraction layers used to organize the system model for electronic and electrical architecture and environment.

including implementation-independent interface definitions, abstract behavior can be included and requirements are distributed. These models are used for analysis and early assessment prior to developing the actual solutions, that is, the design. It is also a means to investigate the principal properties of systems throughout the life cycle, without having to go to the detailed design.

Design Level

The design level contains the detailed specification of the functions and hardware of the EE architecture. Functional interfaces are specified in detail, redundancy is added, and adaptations for efficiency, specific hardware, and middleware, etc. are included. The design level specification of software is equivalent to the final implementation, although it is still a functional description with a certain degree of freedom versus coding, componentization, task allocation, etc. Application software and middleware are separated, but both are represented on the design level, with this degree of detail. The hardware representation contains ECUs, busses, sensors, actuators, etc., as necessary for the specification, analysis, and configuration of functional behavior.

Implementation Level

The implementation level models for software represent the final implementation in terms of code, task allocation, communication/data exchange implementation, middleware configuration, etc. The hardware representation has the same granularity as on design level. More detailed hardware specifications, such as circuit diagrams and VHDL descriptions are not in the scope of EAST-ADL2, and generally not necessary for function/software development.

The EAST-ADL2 relies on AUTOSAR modeling concepts to capture the system on implementation level. "Realization" relations between the AUTOSAR constructs and entities on higher abstraction levels are used to link from implementation to the more abstract entities. This provides the necessary traceability from one or several AUTOSAR entities to one or several EAST-ADL2 entities.

9.5.6.2 Environment Modeling

The environment to the EE architecture, often called plant model, is represented on each abstraction level. The purpose is to capture the assumptions regarding the environment, and to allow analysis and simulation of vehicle functions in their context.

The detail level for the EE architecture increase for lower abstraction layers. The character of the environment model is independent of abstraction level, and is decided by the analysis needs. For example, a simulation for safety analysis may have a very simple engine model, while analysis of a gear selection strategy needs to be more detailed. Thus a set of different environment models can be included to cover different needs.

9.5.6.3 Traceability

The EAST-ADL supports an integrated model containing several parts or (sub)models representing each of the abstraction levels described above. There are links between entities on different abstraction levels and in different parts of the model. For example, "realization" links from components on analysis and design level identify the feature that they realize.

9.6 Conclusion

The complexity and criticality of automotive software and electronics is at a level where adequate support for the management of engineering information is necessary. There are several advantages with introducing an automotive ADL. Most of them relate to the fact that a large share of the engineering information can be contained in the same structure.

At the same time there are challenges with introducing an ADL. One issue is the choice of approach, given the different alternatives described in this chapter. Another is the choice of scope, that is, restricting to only software architecture of individual control units or capturing the requirements, variability, functions, software, and hardware of an entire vehicle. A major difficulty, and also an issue that cannot be solved by mere technical solutions, is the alignment with company internal processes and practices. Linked to this is the user acceptance. This requires education and a thorough understanding of the role of the new approach and tools.

References

1. Schäuffele, J. and Zurawka, T. *Automotive Software Engineering.* Vieweg Verlag, Wiesbaden, 2003.
2. ETAS GmbH. *ASCET User Guide V 5.2*, Stuttgart, 2006. Available at: www.etas.de.
3. www.autosar.org

4. Freund, U. and Möstel, A. Model-based OEM Software Branding of AUTOSAR ECUs. In: *Fifth AUTOSAR Premium-Member Conference*, Brussels, 2006.
5. Thurner, T. et al. The EAST-EEA project—A middleware based software architecture for networked electronic control units in vehicles. In: *Electronic Systems for Vehicles (VDI Berichte 1789)*. VDI-Verlag, Düsseldorf, 2003, p. 545 ff.
6. IEEE. Recommended Practice for Architectural Description of Software-Intensive System. IEEE standard 1471.
7. Object Management Group. Systems Modeling Language (SysML) Specification, ptc/06-05-04, 2006.
8. Braun, P. and Rappl, M. A model-based approach for automotive software development, in *OMER—Object-Oriented Modeling of Embedded Real-Time Systems*, Lecture Notes in Informatics (LNI), Volume P5, GI 2002.
9. Software Engineering Institute. Available at: http://www.sei.cmu.eduSAE International, Architecture Analysis & Design Language (AADL). SAE standard AS5506, SAE November 2004.
10. The TOPCASED project. Available at: www.topcased.org
11. Gérard, S. and Espinoza, H. Rationale of the UML profile for MARTE. In: *From MDD Concepts to Experiments and Illustrations*, Gérard, S., Babeau, J.-P., and Champeau, J. (Ed.), ISBN: 1-905209-59-0, ISTE, September 2006, pp. 43–52.
12. Object Management Group. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP, realtime/05-02-06, 2005.
13. ProMarte consortium. Joint UML Profile for MARTE Initial Submission, realtime/05-11-01, November 2005. Available at: http://www.omg.org/cgi-bin/doc?realtime/05-11-01.
14. Object Management Group. UML Profile for Schedulability, Performance, and Time, Version 1.1, formal/05-01-02, 2005.
15. EAST-EEA, Electronic Architecture and Software Technologies—Embedded Electronic Architecture. ITEA Project 00009. Available at: www.east-eea.net
16. EASIS. Electronic Architecture and System Engineering for Integrated Safety Systems. Available at: http://www.easis.org
17. Advancing Traffic Efficiency and Safety through Software Technology, ATESST. EC IST FP6 project. Available at: http://www.atesst.org.

10

Model-Based Development of Automotive Embedded Systems

Martin Törngren
Royal Institute of Technology

DeJiu Chen
Royal Institute of Technology

Diana Malvius
Royal Institute of Technology

Jakob Axelsson
Volvo Car Corporation

10.1 Introduction and Chapter Overview

Vehicles are being transformed into autonomous machines that assist drivers when accidents are about to occur, inform about traffic conditions, diagnose and upgrade themselves as required, while providing comfort and entertainment functionalities. The evolution of embedded systems technology has provided an important enabler for such new functionalities and improved qualities. In this context it can be noted that an estimated 70% of the innovations over the last 20 years are related to information and communication technologies [3]. Cars are becoming computers on wheels.

The impact of introducing embedded system technology into vehicles has had, and is having, a radical effect on vehicle development, production, and maintenance. Automotive embedded systems have over the past decades evolved from single stand-alone computer systems, simple enough to be designed and maintained with a minimum of engineering, to distributed computer systems including several networks, and large numbers of sensors, electrical motors, and points of interactions with humans. These distributed systems are tightly integrated into the vehicle. They provide flexible information transfer and computational capabilities, allowing coordination among actuators, sensors, and human–machine interfaces (HMIs), removal of mechanical parts, and also completely new mechanical designs. Automotive embedded systems is an interesting area where the mechanical and control systems worlds meet with the general IT world represented by entertainment/telematics functionalities and increasing connections to the vehicle external infrastructure and IT systems. The opportunities are thus enormous, but the new technology also requires new competencies, methodologies, processes, and tools that can handle the flip side of the coin; the resulting increase and change in product and development complexity.

Competition, customer demands, legislation, and new technologies are driving the introduction of new functionalities in the automotive industry. Many new functions in vehicles span traditional domains and organizations. An example of this is active safety systems that assist the driver by receiving environmental information, interpreting the driver intentions, controlling the vehicle dynamics, and, in case an accident is about to occur, informing an emergency center. The increasing system complexity and related increase in costs for the embedded systems development and maintenance create strong needs for systematic and cost-effective development approaches.

Current methods of automotive embedded system development lead to [2,50,70]

- Long turnaround time, since the complete behavior can only be tested in the integration phase.
- Lack of continuity between requirements definition, system design, and distributed system implementation; typically involving different people and with little formalized communication.
- Suboptimal solutions. The organizational structures still mirror the mechanical architecture, and because of a current lack of a systems-level engineering approach for embedded systems design.

The net effect is that systems integration today is a key problem in automotive embedded systems development. To improve this, model-based development (MBD) is strongly pushed in both industry and research.

10.1.1 What Is MBD?

Based on experiences in more mature engineering domains it is well known that the use of models is one essential ingredient in order to achieve cost-effective development of complex systems, see for example Refs. [58,78,85]. Models have always been used in engineering. Models are used implicitly in the mindset of the engineer, in terms of physical models/prototypes and in terms of symbolic/numerical models for the purposes of illustrating and evaluating particular aspects of a system, such as geometry, motion, or heat transfer. The advent of computer-aided engineering (CAE) tools have opened the way for entirely new design approaches where virtual products are created, used for evaluation, for communicating designs with the stakeholders, and as a basis for system realization. Such techniques are already established in mechanical engineering where 3D vehicle models are used to support design, analysis, and visualization (referred to as digital/virtual prototyping or digital mockups), see Refs. [46,74,85].

The use of different types of symbolic and numerical models in a CAE setting, allows properties and alternative designs to be evaluated prior to the development of physical prototypes. Such efforts aim to enable early design iterations at a low cost, thus explicitly supporting early design stages and reducing the project risks. The ability to develop early executable models means that the model behavior can be validated against system stakeholders. This has the important implication that the system context, assumptions, and requirements have a better chance of being captured properly, and that inherent requirement trade-offs can be made more explicit early in the development process. Related to this, the concepts of virtual products and rapid prototyping often enable concurrent engineering in the development, for example, allowing concept evaluation prior to the availability of physical hardware.

It is well known from systems engineering that modeling and simulation are not the end purposes themselves, they should add tractable value. Modeling and simulation are in systems engineering often referred to as a risk-reduction technique indicating their use in supporting engineering decisions. Lessons learned from systems and mechanical engineering also show that all types of models, including mental, physical, as well as symbolic models, have their role and place, but also that computerized models are slowly displacing physical prototypes [78,85].

The discussion in this chapter is focused on computer-aided modeling for embedded systems. In this area there are several disciplines/domains in which MBD is promoted. Examples include "model-based control design"—where control systems are designed based on models of the controlled environment; "model-driven design"—emphasizing graphical descriptions of software and the concept of model transformations; "model-based information management"—where models are used to relate and structure information entities; and "model-based testing"—where models of a design or the environment are used in tests such as in hardware in the loop testing, or to derive test cases. Other related terms include model-based systems engineering, formal methods, and model-driven engineering. These and other interpretations of MBD reflect the consideration of MBD for the purposes of "domain/discipline engineering," such as control or software design, "systems engineering," relating to the overall embedded systems design, or to the "management processes" such as information management.

Several classifications of models, modeling languages, methods, and methodologies have been made that reflect different purposes of MBD approaches [10,20,21,48,51,66, 71]. Examples include distinctions of

- Modeling purposes such as analysis, descriptions, or design based on models
- Levels of formality and types of formalisms
- Different modeling scope and levels of abstraction, for example, referring to the modeling of the system context versus the system structure or behavior, or modeling specifications versus actual behavior

"Formal models," which might also be called mathematical or analytical models, have been shown to be very important tools for clarifying and solving engineering problems. Formal models emphasize behavior and structure for the purpose of predicting system properties. Examples of formal models include geometrical descriptions in computer-aided design (CAD), transfer functions used to describe input–output (I/O) behavior of dynamic systems, and discrete mathematics describing partial orders and finite state machines.

"Conceptual models," most often in graphical form, have proven valuable for communicating and documenting complex software and systems. Examples of such conceptual models include architecture description languages (ADLs) and the Unified Modeling Language (UML) [114]. The formality of conceptual models can vary; for example, enabling syntax and consistency checks but not allowing formal analysis of the correctness of the software logic.

"Constructive models" focus on system operational behavior and form a direct basis for the design itself, proceeding from specifications to detailed solutions. The increase in software and electronics complexity has over the years led to an evolution in the abstraction level at which systems design is carried out. Examples of this include the evolution of programming languages, from assembler to high-level languages, programming platforms, including libraries and middleware, and in hardware design from gate over register–transfer to system level. The result is that system design is specified by constructive behavior models where a central idea is to automate the steps from high-level models to implementation.

All these types of models are strongly related to each other, and are sometimes partly overlapping. Deeply reflected in these types of interpretations and classifications is that of different goals, purposes, and scopes of modeling efforts, and the fact that one single modeling language cannot handle all the aspects of an embedded system. The development of automotive embedded systems requires multiple specialists, each focusing on different concerns, for example, software, electronics, mechanical integration, vehicle dynamics, electromagnetic interference, and safety.

Each community is providing dedicated solutions including tools to solve problems in their area. Architecture and integration then become key problems.

For automotive embedded systems, all these categories of models play an important role. In this chapter we attempt to provide a comprehensive framework for MBD. The framework allows formal models, design models, as well as conceptual models as long as they constitute explicit system descriptions with a documented syntax and semantics, where the models' syntax and semantics must be sufficiently formal to allow computerized manipulation and some level of automated analysis.

Models can be seen as cognitive tools that assist developers in the reasoning and decision-making required in the design process. In particular, the use of models can help to reduce the system complexity as perceived by developers by raising the level of abstraction and providing dedicated views with which systems are described. MBD also supports reuse of earlier efforts, automation of certain design steps, and prediction of system properties. To be efficient, a model thus constitutes an abstract representation of a real or imagined system. To be effective, the models need to retain and bring out the essential nature of this system with respect to one or more explicit purposes.

As with other terms, there are many proposed definitions of MBD. Apart from distinguishing between the level of formality of a model, and whether it is applied for analysis, design, or for communication purposes, models can also be applied to different parts of a system, at different levels of abstraction, addressing different properties and design parameters. These dimensions and different interpretations are elaborated in the following sections.

We interpret MBD as follows: In model-based development, computerized models are used to support communication, documentation, analysis, and synthesis, as part of the system development. In such an approach, the models thus form the basis for the interactions between the teams of the organization, information flow within and between development phases, and for the design decisions made.

A few definitions or interpretations of MBD interestingly include the keyword "driven." Is, can, or should the development be driven by models—and in what sense? One tentative answer is provided from the fields of product data management (PDM) and mechanical engineering. The term model-driven in these fields relates to the use of different types of models including technical product models (to represent different aspects of the product during design such as vehicle geometry and dynamics), product information models (to describe product configurations, parts, and metadata), and development process models (describing development stages, actions, events, and roles of persons involved). When a model of the development process is designed and implemented within a PDM system it can actually drive design, for example, notifying a tester as soon as a design is finished, or causing the software to be automatically rebuilt once a valid configuration can be established. This notion of model-driven engineering thus encompasses several types of product models as well as process models. For the purposes of this chapter we have chosen the term MBD which we find more adequate considering the current maturity of the area.

An MBD approach entails the appropriate use of certain technologies within an organization. Introducing MBD fundamentally requires the presence of motivated people skilled in MBD within the organization and an understanding of the context for adoption including goals, drivers, requirements, and scope. An MBD approach may require changes to the existing processes and to the organization. Equally important is the availability or development of a methodology that provides guidelines on how technology and related underlying theories for MBD should be used. A proper adoption of MBD can help to increase the functional content and product quality, and also assist in making the development more time and resource efficient. (These goals are in conflict. The right choice of MBD approach will enable emphasizing one or more of these goals.)

10.1.2 Chapter Overview

Writing a chapter on MBD of automotive embedded systems is indeed a stimulating and challenging task. Given existing interpretations, practices and a multitude of research efforts, the situation easily becomes confusing. Whereas MBD is firmly established in, for example, automotive mechanical engineering practices, through mature use of tools such as CAD, computer-aided manufacturing (CAM), and PDM, the use of MBD in embedded systems development is still in its infancy. The approaches are fragmented and practices vary across and within application domains. Some of the reasons for this lie in the mechanical engineering heritage and in the evolving nature of applications and technologies for embedded systems. It takes time to change traditions. A comprehensive adoption of models, tools, and methodologies for embedded systems is still far from straightforward. Embedded systems are also multidisciplinary and heterogeneous. Methodologies and supporting tools have had difficulties keeping up with the increasing complexity. This is especially true for software, systems architecting, and integration [4,9,69,82]. The traditions, and sometimes immature methodologies and tools, constitute barriers making the use of MBD approaches. However, MBD approaches at the same time provide means to manage some of the challenges caused by the increasing product, process, and organizational complexity.

One overall goal of the chapter is therefore to provide an increased overall understanding of what MBD is about. The chapter is structured according to the guiding questions, why, what, and how. The framework provided by the chapter is shown in Table 10.1, illustrating the "goals" of MBD, "drivers"—the factors that make MBD worth investing in, and the "means" provided by MBD in order to manage complexity, risk, and reuse, thus improving the chances of reaching the goals. MBD through its means enhances communication, documentation, analysis, and synthesis capabilities, contributing to efficient and effective development processes. An MBD effort will have a particular "scope" in terms of the organizational, process and product "context," and the "concerns" targeted by the MBD approach. The particular context and concerns will determine the "requirements" and constraints, enabling a proper choice of MBD technology to be made.

The chapter complements these viewpoints by providing snapshots from academic state-of-the-art and industrial practices. While the emphasis is on system development, implications on the other life cycle stages are touched upon throughout the chapter. The chapter concludes by synthesizing guidelines for industries that consider to adopt or extend their usage of MBD. From a research point of view, the

TABLE 10.1 Key Considerations in Understanding MBD

Why		What			How
Overall Goals	Drivers	Means	Scope: Context, Concerns, and Requirements	Technology	Keys for Adoption
Functions Qualities Time Resources Innovation	Complexity Criticality Standardization/maturity	Abstraction Formalization and structuring Visualization Refinement Prediction Automation Methodology	Technology/product Processes Organization Business	Languages Models Synthesis Analysis Tools Formats	Defined "why" and scope User and management involvement Strategies/adoption plan Process and organizational considerations

chapter provides insight into industrial needs, and fits the multitude of efforts into a comprehensive picture.

10.2 Motivating MBD for Automotive Embedded Systems

MBD can be approached from many viewpoints. In this section, we first describe the role played by an MBD approach in terms of the basic development activities it supports and the principal means by which this is achieved. We then turn to the main goals and drivers of MBD, justifying the introduction of an MBD approach. The discussion of the MBD means provides a generalized reasoning about potential capabilities of MBD to improve communication, documentation, analysis, and synthesis. The actual achievement of these capabilities will depend on a number of factors that are discussed in the sequel to this chapter. An important point is that the expected benefits of an MBD approach are only potential and depend on the scope, choice of technology, and how the MBD approach is adopted and integrated into an organization.

10.2.1 Role of MBD in Automotive Embedded Systems Development

During the design of an embedded system, models can assist designers in many ways. As the system complexity grows, an MBD approach eventually becomes necessary to support the system design. A central motivation for using MBD is thus as a remedy to manage the system complexity.

The development of automotive embedded systems has to deal with (at least) three aspects of complexity: process complexity, product complexity, and organization complexity. The development process faces several challenging, conflicting, and changing requirements. An example of this could, for instance, be the development of an active safety system providing braking assistance to the driver. The development has to consider requirements on driver comfort, safety, reliability, and performance along with a tight hardware cost budget and constraints imposed by existing functions, components/platforms, technologies, and mechanical design. The different requirements are typically linked to several stakeholders, requiring the establishment of a mutual understanding and trade-offs. The development further involves the coordination and use of several technologies, tools, and activities from multiple domains. Integration among these is essential but challenged by different development speeds (hardware vs. software), tools that do not easily interoperate, distributed information, and tasks that are distributed over different organizational entities. The technical heterogeneity of automotive embedded systems also brings along complexity. The system behaviors are generally nontrivial to predict because of the many types of entities and interactions, and the resulting large state space of the system. The organization aspect is concerned with the integration of resources (humans, tools, information, etc.) from different engineering teams and organizations.

To manage this complexity, MBD approaches can provide designers with support for four main development activities: communication, documentation, analysis, and synthesis of designs. An MBD approach, through its underlying technologies and theories, provides several means to support these activities including abstraction, formalization, prediction, and automation. Based on the support for these activities, it can be concluded that their iterations and combinations, for example for change management activities, are also supported.

10.2.1.1 Communicating Ideas and Designs

Pictorial descriptions of software and systems have been used for many decades given the need to communicate using simplified system descriptions—"a picture can say more than a thousand words, or lines of code." Given the increasing system complexity, the role of pictorial descriptions has become increasingly important [10,14,48,73]. Providing one or more shared system views in terms of models is an important way for communicating the central concepts of a design to other stakeholders. Establishing agreed system models can also improve communication among developers by providing a language and standardized terminology. If the models include an executable behavior description, this enables visualization and thus communication of how a system behaves. This property is known to be very important to support behavior validation with the various system stakeholders—"Is this a desirable behavior of the product?" Executable models can be used for communicating both the desired or expected behavior of a design, and thus also be the basis for analysis and decisions by humans. It should be noted that diagrams or pictures do not always provide the most adequate representations. Many other notations have been proposed including design structure matrices and tabular representations [15,75].

10.2.1.2 Documenting and Managing Design Information

Documentation is of paramount importance for products such as automotive-embedded systems, considering that their life cycle includes development, production, maintenance, and retirement. Proper documentation is essential to support maintenance, system changes, and reuse of already developed system designs/ components. As the product complexity grows it will no longer be sufficient to use text-based documentations. In particular, it becomes very cumbersome to manage changes in documents and the releases of a huge document may delay other parts of the development. It also becomes difficult to handle overlaps and dependencies with respect to other design artifacts—such as, for example, between requirements documents and design specifications (a change in the requirements will necessitate updates or at least checking of the affected parts of the design). Further complexity is given by versions of design entities, and variants of product configurations.* The number of possible combinations of components makes it difficult to have a generic documentation set valid for a complete product line. Concurrent engineering in automotive systems development poses even more challenges since the consistency of information needs to be guaranteed when multiple accesses and changes to the same information occurs [38,50]. Model-based information management, using information models that describe involved information entities and how they are related, can facilitate reuse and maintenance. A model-based approach to information management can further improve reuse and maintenance capabilities by documenting rationales, design assumptions, and the status of design entities. This is possible by attaching constraints, references, and properties to design entities, including "metadata" describing, for example, who is responsible for the design. On the other hand, additional work is required to supply metadata, and a disciplined development process that supports the MBD approach.

10.2.1.3 Supporting Analysis of the System to Be Designed

Analysis can be performed for the purposes of design space exploration, to verify that a system meets stipulated requirements or for the purpose of validation, ensuring that the requirements and expected behaviors are indeed those expected by the system stakeholders. Analysis is particularly important for embedded systems because of their heterogeneity (many entities and types of interactions) and dependability requirements. Examples of properties that are difficult, costly, or impossible to examine by hand include the logical correctness of a system (even smaller embedded systems can have a prohibitively large state space), the timing behavior of a system—especially for distributed systems, and the error behavior of a system (with many sources for errors, and ways by which errors may propagate). In addition, an embedded system is interacting intensively with its environment, requiring that the embedded system is modeled and analyzed together with its environment. MBD can be a very important complement to traditional verification techniques, including reviews and testing, by enabling simulation, rapid prototyping, model-based automated testing, and formal verification. Many of the qualities or aspects of an automotive embedded system are dependent on each other. A change in the system structure may, for example, improve the system flexibility but reduce the performance.

* Variants are here seen as possible combinations of customer choices resulting in a given product configuration at a given point in time, and versions are changes to design entities in stages over time. Configuration management has the role to keep track of versions and variants in system development.

Given a formal system model allows the dependencies between system design parameters and qualities to be explored in a more systematic way, improving the quality of design trade-offs. To support all these types of analyses, system descriptions that capture relevant facets of the system to be analyzed are required, such as required behaviors, function and implementation designs, and the environment behavior.

10.2.1.4 Synthesizing Solutions and Supporting Artifacts

We use the term synthesis to reflect the creation (or composition) of designs and supporting artifacts such as documentation. This creation can be manual or, more or less, automated. A first step of the synthesis is to capture designer ideas (mental models) or other information, such as constraints, in terms of computerized models. Given the formalized model, design can proceed by using the model for communication and analysis, or adding information such that it is documented for later (re)use. Manual creation and manipulation of designs, where new and existing elements are composed so as to form a whole, are supported by modeling languages (with textual or graphical representations) and by tools providing model editors. Automated synthesis is possible when rules have been defined for how models can be created. Several instances of automated synthesis are possible. One example is that of system solution generation, where computer tools are used to search for or derive system solutions—such as a feasible scheduling or an optimal allocation of tasks in a distributed embedded system (see Ref. [6]). Another example is that of code generation from models. Behavioral designs are often developed in graphical modeling languages/tools like Matlab/Simulink [102]. These models can be used to analyze the behavior of functions and later be used as specifications for the corresponding implementation. In many cases, this implementation is carried out by other teams, leaving room for misunderstandings of the specifications. This may lead to discrepancies between design and implementation, and possible faults in the implementation; such problems can be alleviated by use of code generation. Code generation can be used for several purposes and can be applied to the application logic, glue code, or functions part of the platform. Reverse engineering, where models are created to better understand an existing system, for example, from observed behavior to a task behavior model, is another instance of synthesis, albeit still constituting a research area for embedded systems. Other examples of synthesis include generation of documentation, test information, and analysis models.

10.2.2 MBD Means

An MBD approach thus directly supports and has the aim to improve the communication, documentation, analysis, and synthesis in the system development. The principal means with which this is achieved include the following:

- *Abstraction*: Modeling provides the means to define entities and aspects that are useful for design, such as classes of functions (information generalization), failure modes (quality- or aspect-specific abstractions), virtual structures (part–whole structural composition), transfer functions and state machines (abstract behavioral entities), and dependencies (allocation, refinement) although these concepts may or may not have a direct correspondence in reality. The definition of such abstract concepts helps in defining simplified, or rather more adequate, descriptions of the complex real-world in which nonuseful details are eliminated and where important aspects are highlighted.

- *Formalization, parameterization, and structuring*: In order for a model to have a well-defined meaning, and be amenable to analysis and computer manipulation, it must be described using a well-defined syntax and semantics, determining which models are valid in the context of the modeling language, their representation, and meaning. The mappings and relations between several adopted modeling formalisms also need to be formalized. A suitable model structuring is very important in order to achieve readable models and to achieve separation of concerns. Structuring is in turn enabled by the abstraction and formalization. The concept of parameterization facilitates reuse and enables instantiation of already existing models, where different concrete numerical values are assigned to model variables in order to adapt the model for a particular purpose.
- *Prediction*: Through various model analysis techniques it is possible to determine properties of models. These properties may be directly computable based on the model properties (e.g., moment of inertia and logic invariants) or also depend on the model context such as model inputs or assumptions of the platform and other components (e.g., end-to-end response times and the relation between faults and hazards). An overview of analysis techniques is given in Section 10.4.
- *Visualization*: Modeling, through abstraction and formalization, provides means to visualize the system structure. By means of prediction, for example, through simulation, the system behavior can also be visualized (animated), improving the understanding of what the system is (structure), and what it does (behavior).
- *Refinement*: The usage of successive models, that are related through added detail and by including more aspects, is supported through the earlier means including abstraction, formalization, structuring, and prediction.
- *Traceability*: Abstraction, formalization, and structuring provide the means to support traceability of design information. Together with prediction, this also enables investigation of implications of changes, supporting change management.
- *Automation*: The possibilities for automation follow from the other means combined with computer support, enabling automation of all the previously mentioned development activities. Examples include automated initiation of communication to a certain designer/stakeholder upon completion of a development (sub)activity, updates of dependent models when changes have been made in a related model for managing consistency, 24 h/day model-based testing using test scripts, math-based model analysis,

and automated refinement. It can be seen that automation for some of the activities requires models not only of the product, but also of the process—the development activities.

We believe that methodology should be part of the means, as indicated by Table 10.1. However, there is not yet an established methodology for embedded systems engineering, although several pieces are in place. Methodologies for MBD are further described in Section 10.5.2.

10.2.3 Driving Factors for MBD

What are the driving factors for adopting an MBD approach? It is clear that the needs for an MBD approach increase as the products become more complex. An MBD approach provides improved ways of communicating, documenting, analyzing, and synthesizing embedded systems. This is achieved through abstraction, formalization, and structuring of information; prediction of properties; and automation. For simple systems, the tasks of communication, documentation, analysis, and synthesis become easier and so introducing MBD may not be worthwhile. The implementation of an MBD approach is costly and time-consuming since models have to be created, validated, and managed, and the same goes for the acquisition and maintenance of tools, and the required training of personnel.

The ability to reuse modeling and tooling efforts makes the development more cost-efficient. Reuse includes the possibility to reuse efforts invested in modeling, analysis, synthesis, and in setting up tools. The opportunities for reuse increase the more mature and standardized the involved products and technologies become [30,41]. Model reuse, however, requires care to ensure that the assumptions underlying the original model are also valid in the reuse scenario, (see Ref. [33]). MBD supports reuse through formalization and structuring, prediction for understanding system behavior, and parameterization of existing models.

The relevance for MBD is also affected by the "criticality" of the products to be produced, where criticality can refer to cost or dependability in the sense of a mission critical system. While complexity increases the probability of failure, criticality captures the other dimension usually part of risk measures. An MBD approach provides means to support risk management through formalization, prediction, and automation, making analysis of unknown behaviors possible. This is a relevant driver for automotive systems considering criticality at least in the senses of cost, availability, and safety. Model-based analysis and testing allow the system behavior and extreme conditions to be assessed in a cost-efficient way. Such conditions and assessments could for real systems be highly costly, difficult, and hazardous to carry out. A model-based approach moreover provides the benefits of well-defined and repeatable tests.

We consequently identify three main drivers for adopting an MBD approach:

- Complexity management is enhanced by providing support for communicating designs through dedicated models of the system, by managing the documentation and product structures (entities, properties, and intricate dependencies)—providing information integration, and by supporting analysis of product phenomena that would otherwise be difficult or impossible to handle during design. Through models, designers can focus on relevant aspects, and can investigate them more freely, for achieving repeatability and possibilities to manipulate all variables. A model-based approach to information management provides additional support for the reuse and maintenance of different design entities. It should be noted that an MBD approach to some extent increases the development process complexity by adding tools and creating more explicit information. While the prediction and automation means in MBD are beneficial and allow a larger design space to be explored and verified, they also create more information, thus increasing the needs for structured information management.

- Technologies and products that are mature and standardized provide opportunities to apply standardized and formalized description techniques, increasing the possibilities to reuse development efforts.
- Cost or money critical systems benefit from MBD by means to predict and prove system properties prior to their deployment, thereby providing means for risk assessment and management.

We believe that each of the drivers, in isolation, can motivate the introduction of MBD but a combination of them provides even stronger drivers. There are also other factors that affect the introduction of an MBD approach. A summary of these factors is given in Section 10.6. A certain level of maturity with respect to the product technology, tools, competences, and methodology has to be established before MBD approaches can successfully be adopted.

For automotive embedded systems, we find that the drivers in general are strong although they can vary among the automotive embedded systems domains. Given this situation, MBD approaches can provide more efficient processes that can be used to reduce the development time and/or the cost of development. MBD approaches can also be used to target the development efforts to a larger extent toward validation and optimization, yielding products with improved qualities and the "right" features. These potential benefits are further elaborated in the following section.

10.2.4 Potential Benefits of MBD Approaches

We here assume a situation where a careful introduction of an MBD approach has been made by an organization. We consequently assume that the drivers for introducing the MBD approach are strong. If the drivers are weak, or if the adoption of the MBD technology is not carried out in the right way, these benefits cannot be expected. Mismatches between the drivers and the availability of mature MBD tools or established methodologies can explain some of the problems in introducing MBD [82]. Given these preconditions, MBD can provide the following benefits [2,7,48,50,54,58,63,66,69,78,82,85]:

- *Time-to-market.* Increased productivity can be achieved in both direct and indirect ways, where a key ingredient is the provision of more efficient decision making. A direct effect is achieved by allowing concurrent engineering and evaluation of design concepts (e.g., for control, software, or hardware) prior to the availability of a complete product. A direct effect is also achieved by automating development steps, especially those that are tedious, error-prone, and consume lots of efforts, typified by testing, analysis, and model manipulation/transformation activities. Indirect effects are achieved by improved documentation and communication among developers/stakeholders—implying that the work itself can become both more efficient (e.g., faster retrieval of consistent information) and effective (e.g., increased understanding among the stakeholders involved). Another improvement is achieved through early error detection and quality feedbacks, in particular supported through model analysis and early model-based integration efforts. Such efforts minimize the number of iterations in development as well as problems in and after production. A mature MBD approach can thus be expected to reduce the development time compared to non-MBD approaches. On the other hand, faster iterations imply that the gained time could instead be used, for example, to improve the system qualities or include more features.

- *Reduction of various cost categories.* Through computer-aided optimization, MBD can assist in reducing production cost by enabling the selection of more cost-effective solutions. MBD can also increase the quality of the products, thereby reducing maintenance costs. The development costs for automotive embedded systems are currently increasing. The use of proper tools can help developers to better manage the increasing complexity, thereby making the development more cost efficient.
- *Quality assurance and quality enhancements.* As mentioned above, automation can reduce the introduction of faults. The use of models can also have indirect effects by improving the understanding of the system under design—thus allowing the quality assurance efforts to be more efficient. The work of formalizing a system in terms of models can also provide benefits simply by improving the understanding of the system under design, and acting as a kind of review of the system. Enhanced verification is then further made possible through model-based system analysis and testing, helping to ensure the desired system qualities, and in making trade-offs between conflicting qualities. A model-based approach may be mandatory in certain safety critical systems, providing a good ground for a safety case. The system qualities can also be optimized using a model-based approach. Several surveys point to the fact that a main problem in complex systems design is that of requirements engineering. Here MBD can assist in making requirements more easily manageable, communicated, and analyzed.
- *Increase in functional content.* Providing more functions corresponds to an increase in complexity, typically involving more stakeholders, concerns and thus information, analysis, and trade-offs. Such problems are precisely what MBD aims to solve, thus providing support for this.

- *Innovativeness*. There are empirical indications that MBD approaches can also support the innovativeness of enterprises in terms of their ability to produce novel products, functions, and/or solutions [2,57]. It is known from earlier experiences with CAD systems that proper CAE tools can support the creativity of developers, facilitating their exploration and assessment of concepts and solutions. MBD tools can also for embedded systems support rapid verification and validation (V&V), for example, through simulation and rapid prototyping. Empirical findings also support that a proper use of MBD can assist in facilitating communication in multidisciplinary teams [2,49]. Ideally, a system-level information/knowledge management system should support the storing, retrieval, and matching of ideas and solutions, thus improving innovation capabilities.

Modeling requires extra efforts initially to create models unless they have already been created and can be reused, but thereafter makes time- and cost-saving secondary effects possible in relation to the entire development process. Resources and time have to be allocated for these purposes. For enterprises that have not adopted MBD before, an improvement in process efficiency (development time and/or cost) may occur but cannot be expected in the first projects.

To assess the benefits of MBD it is also useful to compare with alternatives common in engineering [30,41,56], including "social design" and "design utilizing paper documentation." While MBD adopts explicit and computerized models, social design relies on tacit knowledge, social networks, skilled and permanent personnel, and the use of physical prototypes and testing. A document-based approach comes somewhere in between social design and MBD. It may also adopt models, but not in a form that is amenable for computer manipulation. The information granularity can be considered to be coarse-grained in that documents provide the granularity with which information is provided. MBD emphasizes the use of formal and computerized models as the carrier of central information. The information granularity can be coarse- or fine-grained, where the fine-grained entities correspond to design artifacts such as individual requirements or software entities. The improved documentation reduces the dependencies on individuals [30].

In a typical and traditional embedded systems development scenario where text documents are used, C-code is written and compiled to a microcontroller. Although such an approach may be adequate for a simple system, the limitations are also rather obvious as the complexity grows. Thus, for this approach to work, the complexity has to be handled another way, for example, by use of a constraining architecture that hampers the introduction of too complex functionalities. An MBD approach handles this complexity in the documentation provided by the models, with tools supporting analysis and integration, and maintaining the whole picture by hiding the detailed analysis in abstractions.

Suitable product architectures constitute an essential ingredient, regardless of approach. The existing automotive architectures have been based on hardware-level modularization, with interfaces at the network level. Such a scheme has proven satisfactory for a long while but cannot appropriately deal with the increasing cross-electronic control unit (ECU)* dependencies, nor provide a basis for product lines encompassing software and hardware. Currently, software architectures are emphasized within the AUTomotive Open System ARchitecture (AUTOSAR) initiative, where a software architecture as well as means to describe and configure software/hardware components are provided [31,94]. This component-based software approach is a step forward but is still not sufficient since functions, nonfunctional qualities, and the system context are not in the scope of the modeling efforts [17,40,81].

An organization thus has the possibility to make a choice between these three approaches. In case a decision is made to go for an MBD approach, there is a need for strategies on how to migrate to MBD from the existing, currently dominating, approaches. In practice, there is also a need for a strategy on how to make the approaches coexist. The mentioned MBD benefits do not come by easily—they do require efforts, strategic choices, and an understanding of the contextual requirements; these contextual requirements are the topic of the following section.

10.3 Context, Concerns, and Requirements

The above sections outlined principal advantages of MBD, drivers for MBD, and how MBD can support development activities. The discussion pointed to the needs to provide a broader contextual perspective to MBD. The usability and efficiency of a particular MBD approach (including the adopted methodologies and technologies) will depend on the particular context in which the approach is utilized, the purposes and concerns the MBD approach is introduced to deal with, and the corresponding requirements imposed by the products, organization, and processes. These are dealt with in this section.

10.3.1 Contextual Requirements on MBD

As an approach to the engineering of complex embedded systems, MBD needs to take the various basic aspects of engineering into consideration. Figure 10.1 highlights the dependencies that MBD technology can have in respect to the business, organization, technology, and process aspects of engineering.

The harmonization among these aspects of engineering versus the MBD technology and methodology is important, and strongly impacts the usability of an MBD approach. MBD technology and methodology can be focused on one or more particular contexts or be flexible enough to be adapted to suit a particular engineering condition. An MBD approach can be directed toward products, processes, or organizational aspects, for example, as an enterprise model with a representation of the structure, activities, resources, people, goals, and constraints of an enterprise. Each area is traditionally dealt with by specialized tools, such as enterprise resource planning and product data management. While the focus of this chapter is on the MBD of automotive embedded systems, employing MBD for the other parts is a natural and

* ECU, automotive term for embedded computer system device.

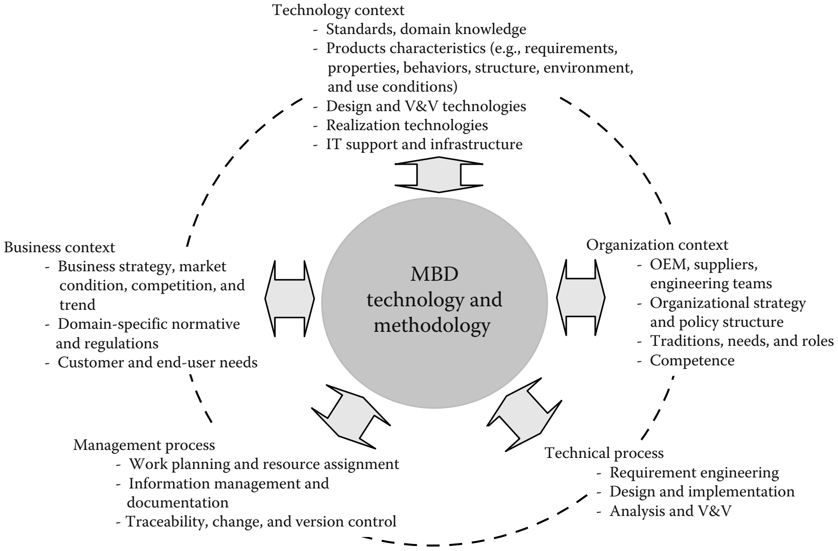

Technology context
- Standards, domain knowledge
- Products characteristics (e.g., requirements, properties, behaviors, structure, environment, and use conditions)
- Design and V&V technologies
- Realization technologies
- IT support and infrastructure
Business context
- Business strategy, market condition, competition, and trend
- Domain-specific normative and regulations
- Customer and end-user needs
MBD technology and methodology
Organization context
- OEM, suppliers, engineering teams
- Organizational strategy and policy structure
- Traditions, needs, and roles
- Competence
Management process
- Work planning and resource assignment
- Information management and documentation
- Traceability, change, and version control
Technical process
- Requirement engineering
- Design and implementation
- Analysis and V&V

FIGURE 10.1 Dependencies between MBD and the engineering context. The arrows from/to MBD indicate requirements on, or constraints imposed by, an MBD approach with respect to its context. The dashed line indicates the interdependencies among the contextual issues themselves.

important complement, enabling parts of the process to be automated—thus coming closer to the concept of development driven by models, and integrated with other parts of the product life cycle.

The requirements indicated in Figure 10.1 are now briefly elaborated, with continued discussions in Sections 10.3.2 and 10.6.

10.3.1.1 Requirements Imposed by the Technology

Technology-related requirements include those imposed by technology used in the product as well as technology used in tools. Each product is characterized by specific properties and phenomena that need to be represented by the models and tools used in MBD. Vice versa, the MBD technology will impose constraints on what can be modeled and analyzed. The diverse functionalities and technologies of the automotive domains pose special requirements on the types of behaviors, properties, and structures to be represented. For example, considering functions for vehicle dynamics control, some form of hybrid systems formalism is required. The concepts of product configurations and product families need explicit support in order to describe parts/functions (their versions), unique product configurations (variants), and rules for establishing valid configurations.

10.3.1.2 Requirements from the Viewpoint of the Technical Process

A technical process consists of activities that make decisions about the intended functionality and qualities, the selection of platforms, the mapping of functions to solutions, detailed design, and V&V. MBD of embedded systems needs to support analysis for the purposes of V&V, for design space exploration, and trade-off analysis, where trade-offs in the design are due to conflicting quality attributes that share common design parameters. A technical process largely stipulates the required modeling, analysis, and V&V support, including requirements on documentation, management, and communication of information. It is clearly beneficial if the documentation, analysis, and synthesis efforts can be reused. In general, each developer or stakeholder will require specific information about the product to be developed—represented as one or more specific models or views. The differences in characteristics of software and hardware are of concern. Software development is characterized by rapid releases and iterations whereas the time constants for development of electronics and mechanical parts are longer. This makes system integration a special challenge where releases from different development teams have to be coordinated. As mentioned earlier, the use of an MBD approach has the potential in improving the situation by enabling early and frequent technical integration through system models.

10.3.1.3 Requirements from the Viewpoint of the Management Process

One common way to control the complexity of products is through the support for information management, implementing the principles of abstraction and separation of concerns. This in turn requires appropriate structuring of the information to relate various engineering concerns like functions, implementations, and their relations. The information management process can and should act as an enabling process for the technical process to be carried out. Model-based information management allows support for communication, documentation, analysis, and synthesis of the product during development. Assume, for example, that a system designer wants to add a new function to an embedded system. To assess where and how this function can be deployed (which ECU, new ECU, sensor usage, etc.), a lot of detailed information is needed. It is the role of the management process to provide this information, consistently and at the right time, to the multiple users in the organization. Information management for automotive embedded systems often has to support geographically distributed and concurrent development. It then becomes important that an overall system definition is available for the purposes of work planning, change, and version control. The reuse and sharing of models requires tool compatibility information, standardized exchange formats, and support for intellectual property (IP) when models are exchanged among organizations. One central issue is how to ensure the consistency of redundant information and to manage the dependency and traceability of information across models.

10.3.1.4 Requirements from the Organizational Viewpoint

Users and organizations have established traditions with respect to terminology, work procedures, and ways of modeling systems. Expectations, traditional roles, and work procedures have to be considered when introducing an MBD approach.

10.3.1.5 Requirements from the Business Context

The business context relates to customer needs, competition, legislation, and market trends. A competitive situation, for example, corresponds to a secondary driver for MBD with needs for more efficient and effective development processes.

10.3.2 Product Concerns Addressed by MBD Efforts

The development of automotive embedded systems involves a multitude of concerns from requirements to implementation, different functions, aspects/qualities, and (sub)systems. The fact that so many modeling languages and tools have been developed reflect the situation that embedded systems design requires many specialists, each having a different viewpoint and requiring specific information to solve their tasks.

Since a model can never be a complete replication of a real system, it is important to have a clear understanding of the product concerns at hand, to determine which information should be detailed and what can be left abstract. Different models will be related to different concerns and thereby have different focus, which leads to different modeling approaches being selected. A particular MBD approach and technology will typically be developed with a particular product scope and concerns in mind, aiming at the following:

- *Targets.* The target delimits what often corresponds to the primary focus in modeling, for example, a subset of the product or integration of product parts or aspects. Integration here refers primarily to model and tool integration with the purpose to support (early) product integration.
- *Design stages.* The design stage is closely related to the concept of modeling at different levels of abstraction. Modeling an early architectural concept, a subsystem, or a detailed component design puts different requirements on the modeling. If a V-process is used for development, this axis is roughly equivalent to the time dimension, but a particular element of the time dimension is that modeling in the early phases must cope with much larger uncertainties than in the later phases. The continued design process is characterized by moving from abstract to concrete system descriptions, in which the precision increases, and the uncertainty and flexibility for larger changes is reduced.
- *Qualities or attributes.* Different system qualities may be in focus such as safety or performance. These qualities need to be defined as requirements or constraints to guide feasible solutions. The qualities are also decomposed into refined requirements. Nonfunctional requirements are commonly refined into functional requirements; one example is safety requirements that often translate into requirements on functions for error detection and error handling. Many attributes represent cross-cutting concerns that go across both functions and modules, including cost, weight, temperature, and electromagnetic radiation. There is thus a need to handle trade-offs between conflicting qualities.

- *Design parameters.* Design parameters refer to the parameters that a designer uses to shape the design. The parameters are related to the "system structuring," for example, determining the number and type of entities and their connections, "system behavior," adopting different models of computation and communications (MoCs), or the "mapping between behavior and structure." For embedded systems, key design parameters include the strategies for mapping of behaviors to the solution structure, execution (triggering, synchronization, and scheduling), communication, and error handling (see Refs. [37,80]).

When adopting an MBD approach, it is essential to define which concerns are to be addressed by MBD. The different concerns provide a multidimensional space closely connected to different scientific/engineering disciplines and stakeholders. Examples of stakeholders and their product concerns are given in Table 10.2. Stakeholder concerns can be characterized using the terms of viewpoints and views. According to Ref. [32], a view constitutes a representation of a whole system from the perspective of a related set of concerns. A view may consist of one or more (architectural) models and a model may participate in more than one view. Many proposals for view-modeling frameworks have been presented (see Refs. [48,77]). The described views have to be understood in the context of the goals for the respective framework. Since the development contexts differ, it can be deduced that one important property for MBD technology is to find a solution that provides the required views or even more preferably, allows the desired views to be defined given a more flexible MBD environment. Developers need to work efficiently with different concerns, but there is also a need to integrate the various models and tools used since they are used to describe

TABLE 10.2 Illustration of Stakeholders, Their Roles, Concerns, and Model Usages

Stakeholder/Role	Concerns	Analysis/Synthesis	Model Characteristics
Electrical engineer/hardware architecture	ECU interfaces and EMC	Electrical load and tests	Logic, continuous, and FEM
Software engineer (body area)	Logics of functionality	Simulation of behavior	Discrete-event
Quality engineer	Reliability	Life-time prediction and FMEA	Stochastic and logic (e.g., failure analysis)
Mechanical engineer	ECU packaging, geometry, and fitting	Cable length and geometry alignment	2D and 3D mechanical
Cost controller	Product cost	Profitability and sensitivity analysis	Economical and uncertainty explicit
Integration engineer	Verification communications and distributed functions	Testing! Automation of tests and generation of test documentation	Discrete-event (test cases) and logical structures (e.g., configurations)
Safety engineer	System safety	FTA and FMEA	Logic, discrete-event, and stochastic
Control system engineer	Performance and robustness, and disturbances	Behavior simulation, robustness analysis, and controller synthesis	Continuous-time, discrete-time, and discrete-event
Thermoanalyst	Temperature	Heat transfer	FEM, etc.

aspects of one and the same system. An information overlap and other dependencies between the models are hence inevitable. Addressing this type of integration and information management is becoming more important as MBD approaches spread within the developing organizations [11,22,23,34,42,48].

The different concerns also provide a way to characterize or profile different modeling approaches [21] and explain why different solutions are found for the classical automotive domains, since each domain is characterized by different qualities/attributes of interest, for example, safety and real time for active safety systems versus logical functionality for body electronics. In some domains, the behavior is predominantly discrete and in others continuous, leading to an emphasis on different types of behavior descriptions.

10.4 MBD Technology

This section provides an overview of MBD technologies including

- Modeling technologies, including languages, models, and relations between models, and between languages
- Analysis techniques, for example, for model simulation and static analysis
- Synthesis techniques, including generation of models and other supporting information
- Tooling technologies, which implement specific modeling, analysis, and synthesis technologies, together with support for design, for example, model editing, simulation and result visualization, model management, design automation, and tools/model interoperability

Figure 10.2 provides an illustration of central parts of MBD technology.

Modeling, analysis, and synthesis technologies are manifested in terms of tools that implement them. The development activities (e.g., analysis) and the concerns of interest (e.g., reliability) impose requirements on the modeling technologies (languages and existing models for reuse) and analysis/synthesis techniques. For example, to support early model-based architectural design, models that can describe the system functionalities, expected behaviors, and solutions at a high level of abstraction are needed. Similarly, analysis at this stage may focus on coarse estimations of cost, system performance, cable lengths, and other relevant metrics. In later design stages, the detailed structure of the software and detailed analysis of effects such as those caused by quantization and end-to-end delays may be of interest. The requirements on the synthesis capabilities will also vary. In rapid control prototyping, for example, the code is typically not required to be optimized, whereas for production code generation, optimization of memory, speed, and accuracy can be an important issue. The analysis techniques in addition impose requirements on models in terms of their information content. An MBD approach typically requires a number of tools to handle different aspects, consequently imposing requirements on the interoperability between these tools.

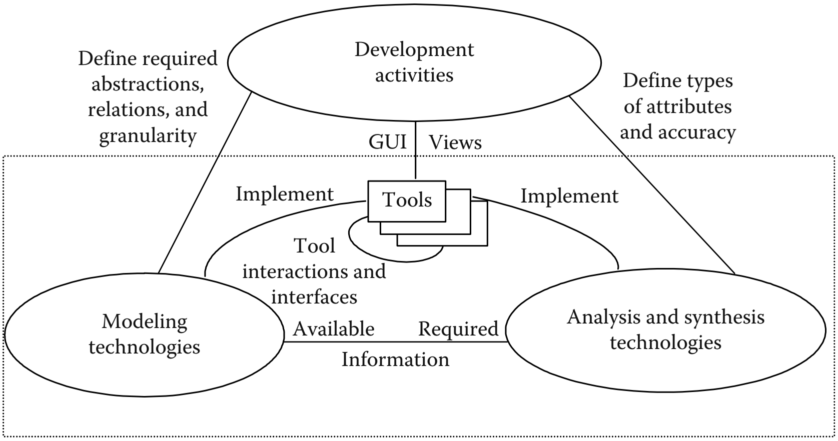

Development activities
Define required abstractions, relations, and granularity
Define types of attributes and accuracy
GUI
Views
Implement
Tools
Implement
Tool interactions and interfaces
Modeling technologies
Available
Required
Information
Analysis and synthesis technologies

FIGURE 10.2 MBD where support for documentation, communication, analysis, and synthesis is provided by modeling, analysis, synthesis, and tooling technologies. The figure illustrates how the involved technologies impose mutual constraints and affect each other.

10.4.1 Modeling Languages: Abstractions, Relations, and Behavior

The purpose in the following discussion is to provide an overview of characteristics of modeling languages, including typical abstractions provided, relations between abstractions and between models, and behavioral models. For surveys and details on different modeling languages the reader is referred to Refs. [4,21,53,69]. Reflecting the broad variety of embedded systems and as illustrated in Figure 10.3, today we find a multitude of programming and modeling languages used in embedded systems development.

Programming languages constitute a particular kind of modeling languages that are focused on providing support for detailed systems design in terms of constructing programs. A program written in C or Java, for example, represents a model since the program is an abstraction of the actual behavior. The actual execution will yield its timing behavior and accuracy of computations depending on the hardware platform

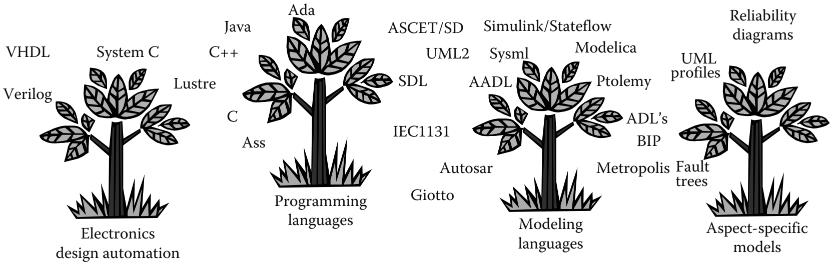


FIGURE 10.3 Illustration of modeling languages available to developers.

(and also depend on the compiler and linked libraries). Programming languages, however, provide limited abstractions when it comes to expressing several of the qualities and attributes of interest for embedded systems [72].

Modeling and programming languages are defined using the concepts of syntax and semantics, and in more detail in terms of an abstract syntax, a concrete syntax, and its interpretation [79]. The abstract syntax defines the concepts, relationships, integrity constraints, and model composition principles available in the language, thus determining all the syntactically correct models that can be built. The concrete syntax defines the form of visualization; graphical, textual, or both. The interpretation defines the meaning of the entities of the language and the resulting models, that is, its semantics. The definition of a language's abstract and concrete syntax is sometimes called its metamodel. A meta–meta model is a modeling language that can be used to define different metamodels. This terminology is used by the Object Management Group (OMG), see Ref. [36], and is exemplified by the definition of meta-object facility (MOF) (the OMG [107]), upon which the definition of the UML2 is based.

From an external point of view, modeling languages can be characterized in terms of the concerns and development activities that they are intended to support. These different purposes should be reflected in the abstractions, properties, and interabstraction relations provided by a modeling language. The abstractions, properties, and relations together define the structural and behavioral concepts that can be captured in a language. In the following, common types found in embedded systems modeling languages are described.

10.4.1.1 Abstraction Types

A number of abstractions can be found in embedded systems modeling languages [21,53]. Typical abstractions include "functions"—representing high-level specification aspects of a system's functionality or logic independent of any implementation technology; "software platforms"—representing hardware units as well as middleware and operating system solutions; "data"—representing information units (signals, tokens, events); "communication"—representing the mechanisms, as well as physical media, for the exchange of information between other abstractions; "system"—representing the complete system, together with its configuration properties; and "generic abstractions" that can be considered as any of the other types of abstractions once specialized or depending on their usage context.

10.4.1.2 Abstraction Properties

These can be divided into structural interface, behavior semantics, and constraints. Structural interface properties deal with an abstraction's organization (what an abstraction is) relating to issues such as size, form, and I/O. Such properties relate to an abstraction's interaction interface with other abstractions. Properties related to behavior define what an abstraction does and how it does it. This includes defining issues such as triggering (event- vs. time-triggered, autonomy), persistence (creation/destruction), timing (e.g., duration), transformations/logic, and storage. The behavior may also be governed by constraints. Another aspect is that of the nondesired behaviors of an abstraction. Any language that supports a general behavior description can use this same technique to describe error behavior. In some cases, explicit constructs are provided for error descriptions.

The viewpoint of time, data, and space provided by a modeling language is manifested in terms of abstraction properties and yield different types of models such as continuous- or discrete-time models. Similar to the treatment of time, data may have continuous value range (or approximated to have this through floating point values) or be discretized. One relevant example of discretization in embedded systems is that of quantized variables, for example, due to limited resolution of sensors, I/O devices, computation, or communication. The combination of time and data can be used to create static or dynamic models.* The space dimension deals with properties that vary in space such as mechanical tension, temperature, or electromagnetic fields. The corresponding properties are described using partial differential equations. In so-called lumped models, the physically distributed property is approximated by being lumped together and is represented by a single variable, for example, a point mass.

A third kind of abstraction property is constraints, which applies to both structural and behavioral properties. Constraints refer to explicit definitions of boundary values, the allowed/disallowed set of values a certain property can take without specifying exactly which value the property will actually take.

10.4.1.3 Behavior Descriptions

Different MoCs are achieved by connecting abstractions with communication and synchronization relationships. The behavior properties of the abstractions, the relationships, together with their semantics, define the behavior of the model. Common embedded systems MoCs include

- Discrete-time models (difference equations), often derived through discretization for the implementation/simulation of controllers or signal processing algorithms in embedded systems.
- Continuous-time models (differential equations). This type of models targets the dynamics of physical systems in the embedded systems environment.
- Discrete-event models (logic and implementation behaviors in computer software and hardware).
- Multitasking models, characterized by platform abstractions which trigger, synchronize, and schedule activities. Preemptions and durations of activities are important characteristics of this MoC.

Many variants of these basic MoCs have been developed over the years [43]. Embedded systems involve several types of MoCs that need to be evaluated jointly, for example, through simulation. Many languages and tools support the definition of several MoCs within the language, for example, Simulink and Modelica [83].

* A static model can be defined without involving time and is thus time-independent in the sense that the output remains the same regardless of the time at which it is studied, given that the input does not change.

10.4.1.4 Interabstraction Relations

Several types of relationships between abstractions are summarized in the following. The relations are in principle valid regardless whether the abstractions are described within the same model/language, or by several models/languages. Many of them define or infer dependencies among models. A special kind of dependency is the case where the same parts of a system are represented using two different modeling languages/tools. As for properties, constraints can typically be defined for the relations, restricting the way in which they are applied. An example of such a constraint could be that applications of different criticality level must not be allocated to the same processor (or task).

- *Decomposition*, referring to the allowed ways of composing abstractions forming a whole, defining part–whole/hierachical relationships between an abstraction and its contained abstractions. Part–whole relationships can be applied to both behavior- and structural-oriented abstractions. In any case, ways are provided for externally accessing the parts of a whole that are not otherwise accessible through the abstraction's interface. This mechanism strongly relates to information hiding and modularity. In behavioral decomposition, the implications on behavior when composing parts into a whole, such as persistence control, ordering and triggering, also have to be defined.
- *Communication*, referring to the allowed ways of connecting different abstractions for the information exchange or physical interaction. Behavior semantics have to be defined for such connections, including protocols and timing. This relation is strongly related to the following one.
- *Synchronization*, referring to the ordering and timing relations between abstractions. Synchronization between abstractions is often realized through communication but other techniques are also possible including the use of pre-runtime scheduling to ensure synchrony between software tasks.
- *Commonality*, referring to the common features between abstractions. Commonality can be achieved through (1) typing where abstract types can be defined from which multiple instances are created, inheriting common properties; (2) common configuration where patterns are repeated across a collection of abstractions; and (3) specialization/generalization where an abstraction type is based on another type, together with some modifications, hence sharing properties that have not been modified.
- *Refinement*, referring to the relationship between different abstractions of the same real system entity. For example, a function refines another by adding implementation details that did not exist in the original function. The term refinement is used in different ways, sometimes referring to refinement that is possible to express within one (and the same) language, and sometimes also to encompass refinements that involve solution decisions and more detailed abstractions, such that refined models may need to be expressed in different languages (referred to as means–ends

refinement). Refinements are closely associated with the design process where implementation details are added successively.

- *Allocation*, referring to the mapping of functions or software abstractions to platform/hardware abstractions. It is a kind of spatial relationship between abstractions and their hardware abstractions, such as software to hardware. The allocation relation involves a decision to map a behavior to one or more hardware components. Once the mapping has been decided, the corresponding function/software can be refined, taking the mapping decision into account (this is the difference between refinement and allocation). A special case of allocation is that of replication, for example, the case where one software task is allocated to several processors causing a redundant solution.

Other dependencies refer to the relationship between abstractions where one abstraction affects the other in terms of a property of one depending on the properties of another, or the existence of an abstraction depending on that of the other, or assumptions taken by an abstraction concerning the other abstraction. An example of one such dependency is a controller whose parameters depend on a particular plant model.

10.4.2 Analysis Techniques

System analysis with MBD relies on tool support to investigate static and dynamic properties of a model, or a set of models. Examples of static analysis include the checking of the compatibility/correctness of interfaces/connections in a model and checking the completeness of a model. Examples of analysis of dynamic properties include simulation of the system behavior and computing the response time of a set of tasks (this property is normally dynamic since the analysis will depend on the timing relations among tasks). Given worst-case assumptions, static analyses are also possible in certain cases.

Analysis of the behavior of dynamic systems can be carried out by simulation or by obtaining analytic solutions. A well-known advantage of simulation is that it has few limitations. At the same time, a simulation approach can only cover a limited number of test cases; deciding on which test cases to run thus becomes an important decision. From this point of view, analytic techniques that provide explicit solutions are preferable. Such solutions are however only rarely available for complex embedded systems behaviors. As an alternative approach, simulation can be combined with partial analytical solutions and search techniques. Examples of the application of search techniques combined with partial analytical solutions can be found in work on allocation and scheduling (see Refs. [6,64]).

Behavior analysis by means of simulation is of great value in achieving a better understanding and in order to investigate alternatives. The possibilities for incremental refinements can be combined with subsequent analysis at each design step. This approach is supported by several MBD tools and is well illustrated by capabilities of embedded control systems design tools that support functional simulation, software in the loop simulation (where all or some functions have been translated into code), rapid prototyping (where models are executed in real time against physical devices to validate models or to test controllers against the real environment), and hardware in the loop simulation (where a real physical controller is tested against a real-time simulation of [parts of] a vehicle), see Ref. [83]. The integration between several MoCs in simulation deserves special attention. There are several approaches to this, for example, by using one of the MoCs to which other MoCs can be reduced and represented [88]. It is, for example, common in simulation to reduce continuous-time models into discrete-time models through numerical integration. A complementary approach is to translate some high-level MoCs into imperative programs (e.g., C) that are easier to integrate and execute in a simulator. In some cases, it may be required to carry out simulation across several tools (cosimulation), each one realizing a part of the system model using different MoCs.

Analysis for embedded systems spans a range of attributes and properties of interest and can be carried out at different levels of abstractions. Different analysis techniques require different model content for capturing the concerns. Simulation and analytical techniques are available for many types of behaviors. Some examples of analysis techniques are listed here.

- Analysis of control functionality provides information about the system input–output as well as internal behavior, and enables to assess performance and stability. The analysis is performed with models capturing the controller as well as the controlled systems' dynamics through differential or difference equations. This kind of analysis is normally performed as one part of function design. Assumptions about controller implementation effects can be incorporated into the analysis, for example, by including time-varying delays, quantized signals, and transient errors [83].
- Analysis of logic and discrete-event behaviors can be used to assess the properties of software and hardware behaviors. Applications include simulation, formal verification, for example, to check that a certain behavior will never occur, and equivalence checking, used to determine if two circuits or programs are functionally equivalent. Formal proofs and model-checking are examples where behavioral properties are (semi-) automatically evaluated on the basis of discrete-event behavior models [4,66].
- Analysis of timeliness and performance provides information about the use of computer system resources and timing behavior, such as end-to-end response times. The analysis requires models that capture application- and platform-level properties, for example, task triggering, durations, communication buffering, scheduling, and synchronization schemes, as well as the timing parameters of system activities and computer resources (e.g., clock period, dispatch time, etc.).
- Reliability analysis provides information about system failures. It uses error models to capture errors in logic, time and hardware. Such models are normally derived from models describing nominal system structures and behaviors [29].

- Safety analysis provides information about the consequences of component/system failures. The aims are to identify hazards, to assess risks, and to support hazard control and risk mitigation. The analysis requires error models capturing the failure semantics and environmental conditions. Classical techniques like failure mode and effects analysis (FMEA), and fault-tree analysis (FTA), in general, require a description of the logical structure of systems due to their focus on the causal relationships of failure events. The analysis can also adopt discrete-event behavior models and corresponding analysis techniques such as model checking. It is common to first capture the nominal behaviors in a formal description and then to augment the description with failure behaviors (i.e., injecting failures in the model) for the analysis (see Refs. [44,67,86]).

10.4.3 Synthesis Techniques

MBD tools provide a range of synthesis techniques depending on the scope of the particular tool. Examples of synthesis support today include generation of models (e.g., code generation from a functional model to code), system definitions or parameters (e.g., the synthesis of priorities of tasks), and supporting artifacts such as documentations related to models. These synthesis examples may include optimization and many involve model transformations, where a model is translated into another model, where the latter may be expressed in a different modeling language. Another case is where documentation is generated from a model—a transformation which is referred to as model to text. The techniques involved in transformations can be classified as declarative and rule based, imperative, or a combination thereof (see Ref. [18]).

One purpose for model transformations is to transform the models of one tool into models that are understandable by other tools, providing information exchange between the tool. An example of this is the transfer of design model information to an analysis tool. In such exchange, standardized formats—treated in the next section—play an important role.

Model transformations in general require more than a simple mapping from one entity to another since the languages may be quite different and since changes to the original model semantics may be required. For example, in the transformation of a design model to an analysis model for verification purposes, it may be required to simplify and change the design model since it may be too complicated for the analysis technique to handle. Another example is that of code generation for automotive embedded systems with resource constraints such as limited performance and memory. The practical implications of these constraints are that the behavior of the original model will be different from that of the generated code executing on a target processor. The model transformation will include making a number of decisions and handling trade-offs, for example between performance and code size [83]. Transformations can therefore be automatic, or partly automatic when human interventions defining trade-offs are required.

10.4.4 Tools

The MBD technologies described previously are incorporated into tools that target specific concerns and activities. Central requirements on MBD tools are summarized in the final section of this chapter. We here discuss tool support for model management, tool interoperation and automation, and standardized formats.

10.4.4.1 Model Management

Tools often come with their own repositories for storing files/models. For information management, there are specialized tools with two main traditions. Software development employs software configuration management (SCM), whereas hardware (mechanical/electrical) engineering uses PDM tools. While most of the general support provided by these solutions overlap, there are variations in the details. A major difference is that the PDM tools to a larger extent emphasize the complete life cycle. SCM has traditionally focused on supporting the management of the large number of source files produced during the implementation phase of software development. While SCM tools support version management based on text files, PDM tools manage versions and variants, and support the definition of information models which allow fine-grained representations of hierarchically structured data. SCM tools, on the other hand, provide facilities for merging software versions in concurrent development [16,22,84]. There are also domain-specific information management tools, for example, for requirements engineering, typically providing some of the features found in PDM and SCM tools.

10.4.4.2 Tool Interoperation and Automation

MBD for automotive embedded systems normally requires a variety of tools, providing different functionalities. Solutions to this may employ direct information exchange between the tools and/or be based on common model management solutions. Model transformations and exchange formats are important to support this. Tool interoperation is supported by tool interfaces and platform infrastructures, such as component object model (COM), and common object request broker architecture (CORBA) [95,96], making (parts of) the tool application programming interfaces (APIs) open to other tools. The APIs can also be used together with scripting features to support automation of design tasks. One trend in supporting tool interactions and modularity is the support for modular tools (exemplified by the Eclipse environment [99]).

10.4.4.3 Exchange Formats and Specification of Data for Exchange

Defining standards for tool interchange is difficult, and there have been many attempts and efforts devoted to trying to define such suitable formats. Today there exist a large number of exchange formats from different organizations such as OMG, W3C [117], ISO [100], and ASAM [92], devoted to different purposes. In general, these standards define a transfer format that allows tools with different internal storage solutions and formats to exchange information. An exchange takes place via a file where internal tool information is translated to and from the file's transfer format. For information transfer, there is a need to define the exchange format and the contents of the file. In many cases, the API is also defined in the standard.

Examples of languages for defining information content include ISO 10303-11 EXPRESS, a part of the STEP standard [110] and document type definitions (DTDs) and eXtensible markup language (XML)* schemas, both standardized by W3C. Examples of exchange formats include various ISO standards as part of STEP and the XML. The XML metadata interchange (XMI) [118], is an OMG standard based on XML. It is used to exchange any metadata whose metamodel is compatible with MOF. The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages. For UML diagrams, the diagram interchange standard is intended to support the exchange of graphical information. Unfortunately, contemporary tools rarely implement this standard, meaning that the support for exchange of UML models between tools—while retaining the graphical information—is currently inadequate.

Many specific standards are based on the above-mentioned ones. For example, the ASAM FIBEX format, a standard for the representation of network data for automotive systems, uses XML and XML schemas. This is also the case for the new automotive software/hardware component description language developed in AUTOSAR, and the recent requirements interchange format (RIF) [109], a format for the standardized exchange of requirement information intended to replace text documents.

10.5 State of the Art and Practice

This section provides snapshots from industrial practices of MBD. The presentation is exemplifying but is intended to be representative in that it includes examples from several domains, from original equipment manufacturers (OEMs) as well as subsystem suppliers, where MBD is used for different purposes. MBD of embedded systems is a hot research area. Following the snapshots from state of practice, we give a brief overview of some of the active areas concerning MBD-related research. The interested reader is referred to the following references for further reading on these topics [3–5,9,42,69,79,82].

10.5.1 Automotive State of Practices

In general, development practices vary to a great extent over different companies, and across the automotive domains. This is not surprising because of the heterogeneity of the automotive embedded systems. While some domains are characterized by model-based approaches that support several steps and aspects of the development, other domains still mainly rely on written documents for specifications and handwritten C-code for software development. In general, there is a limited use of system-level MBD tools, for integration/networking, systems architecting, and

* XML, standardized by W3C.

information management. A complete development chain involves several tools and pieces of information that are loosely integrated. Examples of problems in this area include efforts trying to combine function specification tools with tools for software design and with specific engineering analysis models/tools (e.g., for safety and reliability).

10.5.1.1 Model-Based Cross-Enterprise Communication and Integration

A typical automotive OEMs development process for embedded systems includes writing specifications for subsystem suppliers, and then testing and integrating the resulting ECUs. In other cases, the OEMs may write parts of the software for the ECU, where integration can be performed by the OEM or the subsystem supplier. For a few ECUs, the OEM will develop the complete application software, often along with a proprietary software platform. The specification and integration of separately developed subsystems requires that the partners agree at least on the component interfaces, nonfunctional properties (e.g., weight), and the procedures for fault reporting and diagnosis. In current practices, a widely used method for interface specification and matching is through the use of standardized networks, mainly controller area network, providing a shared communication format and protocol among partners. The communication between partners is then based on text-based documentation, complemented with direct communication for consensus and conformity checks. As such an approach is restricted to the network level, no direct support is provided for early analysis as well as for the interface matching of fine-grained components, such as when external functions and software components need to be directly integrated in a product or its subsystem. Several efforts and newer tools attempt to support system-level design (see Refs. [98,113,115,116]). Some OEMs and suppliers have tried to solve the problem by using Simulink or other models to improve information exchange. One limitation is that in most cases the approach is delimited to the functional aspects. Issues that often remain open include, for example, the traceability of technical decisions and hypotheses, and the interface semantics with regard to timing, synchronization, safety, and variability. Moreover, the concerns of IP can also cause problems in conformance checks and behavior composition.

10.5.1.2 Model-Based Information Management

The problems today when introducing structured information management in industry are largely of organizational nature. There are strong needs to support the transition from text-based to model-based information management. Various approaches to information management are practiced by different domains and disciplines, based on PDM, SCM, or tools with similar features. This is also an area where several new tool vendors are appearing. In practice, however, model-based information management in the automotive industry of today is still a vision, not only at the systems level, but also for many domains and disciplines. The majority of the developers in studied companies are still defining their functional requirements in text-edited documents. The need to go through thousands of documents tracing changes is a tiresome but everyday reality for many designers. Although the ambition to transfer into a model-based paradigm is prioritized in many organizations, many others do not recognize this effort. The problems are common and of the same nature: overflow of information, problems with version management, outdated information and late testing, though they are tackled differently. Instead of focusing on a model-based approach, some organizations choose to find a mutual definition of how to write requirements as one measure to integrate the information. The information overflow that follows from complex product development is especially challenging for electrical/electronics and software engineers. Here the need to integrate and balance requirements from different engineering disciplines is a prerequisite for quality product development.

10.5.1.3 Model-Based Vehicle Motion Control Engineering

For control systems development in the automotive industry, MBD is in many areas already a standard design approach although the extent varies between different companies and subsystems. Characteristic for mature adoption is that a model-based control engineering approach is used, where models of the vehicle and the control algorithms are used. Typical application areas include chassis systems, power train, and climate control. One alternative to the model-based approach is through tuning, in which the control behaviors are primarily determined by tuning. For example, the design of engine control is an automotive domain with little tradition of model-based control, relying heavily on lookup tables and calibration.

MBD supported by CAE tools (see Refs. [91,98,102]) facilitates the design of advanced control functionality by allowing incremental development with early V&V before the mechanical and electronics hardware are available. A typical design starts with a behavioral model that is used in, and refined through, simulation, rapid-prototyping, software-in-the-loop simulation of application software, production of target code, and finally with hardware in the loop simulation, where the set of ECUs are tested against a real-time simulation of the vehicle and nonavailable ECUs. In some companies and domains, the control systems development is not connected to the embedded systems implementation. In this case, the control system model will be part of the specification for the implementing team. The tool support for the model-based approach is today mainly limited to a per ECU basis. The integration of application software to the system software in one ECU can be done manually or using tools that are more oriented toward software development, such as UML-based tools. In this case, the control system code is integrated with the system software, just like any piece of hand-written code.

10.5.1.4 Model-Based Generic Functionality and Software Design

In a vehicle, both the body and infotainment domains consist of functions that are mainly discrete in their nature. Such functions normally have complex logic interactions and can be sensitive to synchronization, performance (e.g., bandwidth and throughput), and usage conditions. To enable early verification, different state-based modeling and analysis techniques as in generic software systems have been used in current practices. For example, the creation of early models of vehicle HMI, either as desktop simulations, in driving simulators, or mounted in an old vehicle, is an important practice to validate such interface functions with regard to performance, usability, and user-friendliness. With respect to software modeling, the use of UML has been increasing. However, the use of UML is not widely spread for automotive embedded systems. Subsets of UML are used for a number of purposes including communication and documentation. There are also cases where the use of UML tools includes code generation.

10.5.1.5 Model-Based Testing

Model-based testing is becoming increasingly utilized where several modes of testing are provided by the use of models and tools, from pure model-based simulation, over software simulation, to hardware in the loop simulation. A challenge for model-based testing is the need for system configuration management and for defining relevant test cases—calling for a more systematic approach to embedded systems development.

10.5.1.6 Model-Based Safety Engineering

The overall aim of safety engineering is to identify hazards, to assess risks, and to carry out hazard control minimizing the risks. The analysis normally requires an understanding of possible component errors (i.e., in terms of failure modes) as well as their propagations within a system and their consequences. For automotive systems, a set of classical safety analysis techniques like FMEA and FTA are used in engineering practices. These techniques rely on analytical models capturing the error logics of a system (i.e., possible errors and error propagations). One challenge is that the error models as well as the analysis outcomes are often kept separately and may diverge from the actual design [60].

10.5.2 Research and Related Standardization Efforts

The heterogeneity of automotive embedded systems, in terms of different domains, characteristics, and properties, have given rise to a very large number of modeling formalisms, techniques, and tools to support MBD. Individually, each of these technologies only covers delimited system concerns. The fact that these technologies are not always integrated combined with a lack of a common terminology and understanding, can cause severe problems. There is a large number of research topics in the area of MBD, including modeling languages, model integration and management approaches, methodologies, specific techniques for analysis, system refinement, generation of test cases from system specifications, optimization and synthesis techniques, reverse model engineering where models of existing software/hardware can be created in a systematic way, modeling reconfigurable systems, HMI for representing complex systems, component models, and platforms. Several of the topics are also interrelated. A challenge in following this evolving field is due to its multidisciplinarity; the topics are treated in a number of fields including systems engineering, embedded software, electronics design automation, reliability and safety engineering, control engineering, mechatronics, automotive engineering, and more—and thus at corresponding conferences and journals. In the following, we focus on the first three above-mentioned topics, emphasizing efforts in the direction of embedded systems engineering.

10.5.2.1 Modeling Languages for Embedded Systems

As illustrated in Figure 10.3, a large number of languages already exist, or are in development for embedded systems. Here we focus on two categories of such languages, namely ADLs and behavior modeling languages. To support design and systems integration, there is a strong need for standardized descriptions of embedded systems—this has spurred the development of ADLs. A typical case is for a systems integrator who should specify and integrate a large number of components (functions, software, electronics, sensors, and actuators). A systems description language can enable communication, documentation, and coordination of such integrated systems, as well as formal analysis and automatic system synthesis. A typical case for synthesis is that glue code/logic is generated, for example, to support the interactions between software components. The emphasis for ADLs is on systems structure, possibly at different levels of abstraction. However, some behavioral aspects are usually also provided to enable systems-levels analysis. There are several examples of relevant ADL efforts including AADL, AUTOSAR, CAR-DL, EAST-ADL, and related OMG languages and efforts [1,13,17,87,89,93,94,111]. We do not explore these efforts much further in this chapter, since there is a separate chapter devoted to the topic of ADLs—to which we instead refer the interested reader.

Behavioral modeling languages share the desire to provide high-level behavior abstractions, above software/hardware implementation level. Further commonalities include the desire to capture many different types of behaviors (models of computation) and properties of embedded systems, and the development of transformations or refinements, whereby the abstract models can be translated into software or descriptions from which hardware can be synthesized. Languages in this category typically support behavior simulation, analysis, and synthesis. A research challenge is to support multiple MoCs within the same framework while providing a basis for sound analysis and synthesis. A related research challenge is to support the refinement of behavior models to distributed embedded computer systems. There exist several commercially used languages in this category including Simulink, Modelica, and SystemC [102,106,112]. Examples of research approaches include Ptolemy, Forsyde, Milan, Metropolis, and other codesign efforts [68,69,83,104,108]. We describe below Metropolis as an illustrative example of research efforts in this category. The connections and/or integration of ADLs with behavioral modeling languages constitute one interesting research challenge.

While UML1 and UML2 lack many properties required for modeling embedded systems, there are several OMG developments that try to address such issues, complementing the UML2 norm. One way to deal with such extensions is to define a UML2 profile using the available extension mechanisms of the UML. The EAST-ADL effort is one example where the UML2 language is being tailored to a particular domain and more formalized, by the use of UML2 extension mechanisms. A current OMG request for proposal, MARTE, is addressing this, in order to define a new UML profile for real-time embedded systems, adding properties for specifying timing requirements and component behavior, for example [101]. Another OMG effort is that of the recent SysML standard—a visual modeling language for systems engineering applications [111]. SysML supports the specification, analysis, design, and V&V of a broad range of systems and systems-of-systems. It is implemented as a UML2 profile, adding some diagrams and constructs to UML2 (mainly regarding requirements and parametric associations). SysML provides an interesting framework that still needs to be proven for the area of embedded systems.

Metropolis is an environment for the development of embedded systems from specification to implementation with a "platform-based" methodology and successive refinement [69,103]. The approach stems from the area of electronics design automation and the idea is to investigate if proven concepts for hardware synthesis can be extended to embedded systems. The aim is to meet the increasing needs on complexity control and verification, and to provide well-defined semantic links between specification and implementation. The focus is on analysis and synthesis through formal methods as well as on the integration of heterogeneous models. Metropolis provides a metamodel of computation that forms the basis for expressing commonly used MoCs. System modeling in Metropolis includes abstract behavior specifications, architecture, the mapping of the behavior onto the architectural elements, and the expression of constraints over quantities such as time or energy. In the architectural model, computational and communication components are characterized by services and costs, such as energy. Mapping corresponds to refinement of behavior that provides emerging properties. The approach explored in Metropolis is to support design optimization combined with analysis and verification tools.

10.5.2.2 Model Integration and Management for Embedded Systems

As described earlier in this chapter there are many possible relations among the different types of tools that are used in embedded systems development. These dependencies and development processes can be used to identify different integration patterns [22,34,76]. A typical example where integration is required is the need to support several types of analyses. Since the tools and analysis techniques are fragmented there is a need to develop support for extracting and mapping information of heterogonous types. The integration can be achieved through common formats or metamodels together with support for tool interfaces and interoperability.

One approach to handle the information is to adopt a PDM approach where the information is stored either centrally or in the domain tools. Regardless of the approach taken, it is important to realize that models have components and structures that need to be versioned, configured, and documented. A challenge in doing this is the handling and integration of the disparate sources of information. A resulting challenge is to provide management tools that support the full range of embedded systems development and that allow fine-grained information management. Product information models are important as a basis for management. Today there are standardized models in mechanical engineering but not for embedded systems. Advances in embedded systems modeling languages may provide the foundations for establishing standards.

Examples of related research efforts are given in the following (see Ref. [11] for more).

- *GeneralStore* is a platform enabling a development process running from models to executable code, in which heterogenous CASE tools (e.g., Matlab/Simunk and ARTiSAN RT Studio [90]) and their associated code generation facilities are integrated. UML models are used for overall system design, and subsystem models can be described in the discrete- and time-continuous domain. This is achieved by providing metalevel definitions of CASE data in UML, and then integrating the metamodels in a MOF object repository. The data interchange between CASE tools is supported by XML. GeneralStore provides support for configuration management [65].
- *ToolNet* is an integration platform, managing the integration of domain tools targeting specific design phases or aspects of embedded software systems, such as DOORS [97] and Matlab/Simulink. This approach leaves the domain data at their respective tools. Data integration is achieved by specifying a virtual object space in terms of relationships and consistency constraints of domain tool data, which is then stored and managed in a relation repository. Standardized APIs are used to support tool access and XML-based export of tool data. ToolNet provides a graphical user interface for navigation [25].
- *Model integrated computing*. In research at Vanderbilt University, methods and a prototypical tool-suite for MBD including model integration and management has been developed. The approach emphasizes the use of domain-specific modeling languages and model transformations for integration with different engineering tools. The transformations are implemented through a mapping of the metamodels of individual models. To support exchange of tool data, different tools are connected on the basis of CORBA/COM technologies. While each tool has an interface adapter that provides the data access (e.g., through a data file or COM interface) and syntax manipulation, the system provides a central semantic translation service for preserving the semantics of tool data being exchanged [35].

10.5.2.3 Methodology Support for MBD

Methodologies that support embedded systems MBD are required. As part of the methodology there is a need to define how models should be used in the development process and how different types of models and modeling languages are related to each other. A methodology should provide guidelines on how and when to deal with different product phenomena. A key challenge is the development of methodologies that support cost-efficient and systematic design and verification supported by models and analysis technologies. Integration with the automotive mechanical engineering faces several challenges on the path toward systems and mechatronics engineering. Today, software and electronics are often seen as being late in the product development process, though their lead times are perceived as shorter. While the subsystems and components are optimized, this is not the case for the complete system.

Research and development of methodologies is an interesting area which should draw from experiences in systems engineering, concurrent engineering, and related disciplines [8,12,27,58,61,62,78,85]. A common assumption in these approaches is that a properly managed process leads to satisfactory products and efficiencies. On the other hand, while emphasizing management aspects of engineering, all these approaches assume that the products as well as necessary domains knowledge (e.g., automotive engineering) is described and provided [24]. Properly capturing such information is particularly important in order to manage various concerns, to assure product qualities, to obtain a consensus among stakeholders, to improve the processes and plan resources in organizations. For example, to allow concurrent engineering, it is necessary that the product information and expertise traditionally associated with a specific development stage are made transparent and available at every stage of development. These approaches also need adaptation to take software and embedded systems properties into account. Similarly, methodology development should draw from existing software engineering processes [39,52], where, however, adaptation and considerations of embedded systems characteristics require further work. There are several pieces and contributions toward methodology for embedded systems MBD in the literature (see Refs. [4,10,19,28,42]). One example is that of platform-based design, a methodology derived from electronics design, where the conceptual idea is to provide reusable platforms at different levels of abstractions. The platforms constrain the design and at the same time constitute a kind of product line at each level of abstraction. A challenge remains how to support formal refinement between different abstraction levels and in integrating models that describe different system aspects [69].

10.6 Guidelines for Adopting MBD in Industry

The introduction or extended usage of tool chains supporting MBD is not unproblematic. Introducing tool chains causes a reliance and dependence on particular tool vendors and requires training of personnel. An understanding of the scope (providing adequate requirements for technology selection), the presence of motivated people skilled in MBD within the organization, and the consideration of strategic issues belong to the prerequisites for the successful usage of MBD.

This chapter has described several issues that are of central importance to MBD. This section attempts to summarize and elaborate these issues with the aim to provide guidelines or at least checklists for considerations that are central for a successful adoption (recall Table 10.1).

Relevant questions when considering the use of MBD include the following:

- What are the goals and drivers for MBD in our setting?
- What is a suitable scope for MBD in our organization with respect to processes, stages, activities, product concerns, and business context?
- What are the implications (opportunities and challenges) with respect to existing formal processes, informal work practices, and which organizational units will be affected?
- What are the resulting concrete requirements on MBD technologies, which MBD technologies are relevant and what are their limitations?

- How can model/information management and integration be solved and what should the corresponding IT architecture look like, taking into consideration future evolution?
- How to deal with legacy information, models, and tools?
- Considering the above, what steps need to be taken to appropriately introduce and maintain the new development approach?

The involved efforts depend on the scale and scope. MBD approaches that cross discipline/development stage/domain/enterprise borders are likely to require more effort. Considering current practices that are dominated by localized MBD efforts, many new efforts concern systems-level MBD, aiming to integrate MBD efforts. Systems-level efforts constitute drastic changes that require several strategic considerations that will be discussed in the following. The primary concern of this section is that of systems-level MBD efforts, but we believe that many of the issues raised are also valid for smaller scale MBD efforts.

In the following we discuss strategic issues, processes, and organizational considerations when adopting an MBD approach; desired properties of MBD technologies/tools; and additional hints on arguments and pitfalls. Some of the strategic issues are further discussed in later sections.

10.6.1 Strategic Issues

Just as any other investment, the introduction of MBD has to be motivated. However, there is no common method to evaluate the effects of such an investment. We believe that the framework provided in this chapter (Table 10.1) provides a first step in order to make a strategic choice of when and how to adopt MBD. A further baseline is formed by developing the fundaments of the MBD effort, after assessing the goals, drivers, and scope. The fundaments include the definition of a product ontology that defines product parts, relations, and properties, and a related reference process with stages, activities, flow of information, and events. It is important for the process model to clarify the stakeholders and views associated with different roles and concerns—compare with Table 10.2. We recommend that these fundaments be described as models themselves. The models can be compared with the current processes and organization in order to better assess the potential changes required. The ontology, or information model, should focus on what is considered the most relevant parts for the MBD effort, while exploring the interfaces to other parts (processes, product, information, organization).

Further strategic issues related to the consideration of a larger scale MBD effort include

- *Planning for the MBD adoption and long turn evolution.* A plan for adopting the new or modified MBD approach needs to be established. A new development approach such as MBD requires raising motivation among users and managers, and establishing competence and a new culture in the organization. Education and the development of pilots, making the case for MBD, are important for this. A given organization will most likely already have MBD approaches in place for some domains/disciplines.

Other domains/disciplines and the overall embedded systems engineering, in particular model management and integration, are however not likely to be based on models. In the planning for an MBD approach, model integration, management, and the handling of legacy information and artifacts becomes central.

- *Developing a suitable information and tool architecture.* This issue, in particular, requires that dependencies between development activities and models be investigated. Given this information it is easier to address different solutions for model management and model/tool integration, and also makes it easier to address parts of the first strategic issue.
- *Technology/tools selection.* An important starting point is to define the organizations' needs for functionality in MBD tools. Once this is done, a mapping of the needs to the capabilities of existing modeling tools can be made. Normally, there is a large resemblance between the tools but in complex development it is hard to find standardized tools that match the company's needs exactly. In the area of MBD for embedded systems there are also some areas that are as yet uncovered by commercial tools. The choice and trade-offs between standard, tailored, or proprietary tools, thus becomes an issue.
- *MBD culture.* One major challenge is to adapt people's mindset from written text documents to MBD. In order to achieve MBD, a new way of thinking that oozes through the organization needs to be obtained. For MBD to actually be acknowledged in a development setting, people should feel the need to read, retrieve, and store information through the use of models. Since information is used differently by engineers from different design contexts, it is important to understand what other designers from external engineering disciplines need to take part in the domain-specific information.
- *Adapting tools/processes to user needs or the users to new processes/tools.* This issue will depend on organizational traditions and to what extent the current processes and established work practices conform to the desired MBD processes.

10.6.2 Adopting MBD: Process and Organizational Considerations

In the adoption of MBD, it is essential not to neglect organizational aspects [2,55,57]. In the following, we discuss issues related to MBD development processes versus work procedures, and the pace and approach for introduction. For more reading in this area, see Refs. [26,45,47,59].

10.6.2.1 MBD Development Processes versus Work Procedures

It is known that developers establish their own informal work procedures and adapt the formal product development processes to fit their everyday work situation. When introducing MBD tools for technical or information management activities, the question that arises is whether standardized product development processes should be used for MBD or whether the processes should be adapted to the informal work procedures used in the organization. Thus, another organizational prerequisite that ought to be taken into consideration is to what extent the need for standardized tool solutions out-levels the need to avoid forced adaptations made to the work procedures. If the user-perceived needs are not fulfilled by the adopted MBD tools, developers are forced into unwanted work procedures.

If the integration need is strong, possible technical as well as terminological interface problems have to be overcome. One challenge when introducing MBD into automotive organizations might be the lack of a coherent terminology. A possible outcome of an adaptation to MBD is the need for changed or new organizational roles due to new work tasks, for example. It is also possible that the organizational terminology differs between different engineering disciplines. In the case of introducing information management systems that manage information from different models, the use of a (coherent) terminology needs to be defined. This can be seen as an advantage of systems-level MBD approaches, since they require organizations to focus on these fundamental aspects. However, the importance of defining how the information should be structured must not be underestimated. Often a formalization of requirements specifications is necessary when MBD is introduced. Even though tools for MBD are available, the tools themselves do not inform what the proper abstraction level for the information should be. A methodology, adapted to MBD and taking work procedures into account, should define how models are to be used in the development process and how different types of models are related to each other. As a complement to methodologies, there is also a need for concrete guidelines that assist users of MBD technology. There are, however, very few publicly available guidelines. A related effort is that of the MISRA guidelines for C programming [105]. As a consequence, many companies who have adopted some form of MBD have developed their own proprietary guidelines.

10.6.2.2 Pace and Approach for Introduction

Another aspect that needs to be considered is at what pace an introduction can be made. A step-by-step introduction process or a full rollout can be applied [26]. The importance is to find incentives for presumptive users of modeling tools, to motivate and educate them [49]. Individuals in an organization have different levels of receptiveness to change. When introducing MBD it has to be decided whether to take a top-down approach or, in cases where it is possible, to allow a bottom-up approach. The latter evolves from the designers' needs and highly motivates use of new tools among designers. A bottom-up approach thus involves designers from the beginning and focuses on the individual's needs and adaptation to current work practice. However, when such bottom-up approaches are not anchored or actively supported by top management the tooling solutions tend to spread to separated "user islands," where system adaptations are made along the organic growth [49]. A top-down approach concludes what information needs to be stored and shared from a more global point of view. For domain MBD tools, a global approach also considers interactions between domains (models, tools, and people) and how the system requirements, functions, and overall V&V are performed.

10.6.3 Desired Properties of MBD Technologies

It is obvious, but not always the case in practice, that the technology requirements should be based on tool functionalities that are needed by the organization, not on what functionalities are available on the market, as delivered by different tool suppliers. The requirements on MBD technologies as manifested by tools will be determined by the particular context where MBD is adopted; the properties of the system that MBD will be applied to determining the technical scope, the goal—for example, to improve the product quality or time to market—and the MBD means and activities that are relevant to achieve the goals. Other requirements follow from process and organizational constraints such as concurrent engineering.

Here, we divide the desired properties into three kinds of requirements: general requirements, technical process and product requirements, and management process requirements focusing on how to handle several users and lots of information.

The general requirements include:

- *CAE tool interoperability*. An MBD approach will involve a set of interrelated tools that in turn will interact with several users, other already existing tools, and information/models. Well defined tool interfaces (external as well as internal) are thus important. The direction should be to strive for as few tools as possible, where the solution will have to be a trade-off between the capabilities provided for different concerns and the efforts on model/tool integration. The larger the scope of an MBD effort, the larger the implications of model management and integration will be. One specific aspect is that of dealing with overlapping/duplicated services and information between the tools.
- *Computer–user interaction*. Suitable computer–user interaction techniques need to be provided between the tool and the users. In particular, the usage should be intuitive and provide feedback such as consistency and correctness checks. Single/multiple users as well as the support of different usage conditions (lab, in-car, etc.) have to be considered.
- *Adaptability*. A given technology and set of tools will require some adaptation to a given context, and to account for changing requirements/contexts. It is therefore important that the services are configurable to some extent. This also provides means for later adaptations such as the incorporation of new tools or changed work procedures. A common requirement is that the tool services have an open API including access to models inside the tools. For information management systems it is important that the information model be customized. It is preferable if the tools support standardized exchange formats for both inputs and outputs.

- *Cost issues*. The cost efficiency associated with MBD technology is important including issues such as licensing models, and the availability of adequate support and training.
- *Performance and scalability*. To further support efficient development, the models and tools used need to provide the required performance. Model assembly time and execution time have to be reasonable. Here, the trade-off between accuracy of results and speed of results has to be dealt with. The tools must be able to handle systems of the required size for the intended MBD tasks.
- *Dependability*. With MBD, the system design will rely on various technologies implemented in terms of tools. It then becomes important for these tools and technologies to be dependable themselves. There are several aspects of dependability, including the availability of several users, user references, guidelines, and methodologies associated with the tools. A related aspect of dependability is the trust and reliance with respect to the company that provides the tool. The need for verification of the involved tools and their model manipulations increases for dependability-critical applications. For these purposes, it is beneficial if the tools provide both analytical, simulation and real-world testing capabilities, allowing model behavior to be compared between different environments, and also supporting model validation. For verification of code generation the ability to trace from the model to the code and vice versa is important.

The technical process requirements include:

- *Modeling languages*. The desired properties are determined by the extent the scope of modeling. It must be possible to represent the desirable properties, structures, and behaviors of the targeted product/subsystem in a useful way. The usefulness relates to the expectations and experiences of the organization. For example, the need to support related design activities, such as analysis, creates additional requirements on the language formality and information content. Additionally, libraries of model components supporting design should be supported. Given increasingly complex systems, the system models also become large. Structuring support for models is thus very important.
- *Communication*. Support for visualization of execution runs and results are central. The provision of different views, showing different aspects of the models, facilitates communication among multidisciplinary experts.
- *Analysis*. For a given scope, an MBD technology needs to support the required types of analysis (e.g., fault propagation or power consumption), preferably both through simulation and analytically. Since the analysis is done at the level of the model, model-level analysis (static and dynamic) capabilities should be provided by the tools. Analysis including real-time simulation poses special requirements on the code generation, the simulation platform, and its interfacing capabilities.

- *Synthesis*. The provisioning of synthesis in terms of generation of code and documentation can be highly useful in reducing manual translations and possible manual mistakes. The requirements for synthesis include tailorability, transparency, and verification (see dependability). Tailorability involves issues such as optimization (e.g., memory, performance, or accuracy in code generation), the style and naming conventions of the generated code. Transparency means that the rules of the synthesis should be well documented. Adaptation of the involved transformations could be achieved indirectly, through tuning directives or by granting users direct access to the underlying model transformation rules.
- *Automation*. The requirements to support automation of analysis or other tooling facilities include, for example, the provisioning of an open API and scripting facilities whereby the automation activities can be described. For advanced automation, the scripting can become difficult, and may itself need model-based support for communication, documentation, analysis, and synthesis.

The management process requirements include:

- *Multiple views and model integration*. The provision of dedicated system views promotes complexity management. The requirement to support multiple views is linked with the requirement to manage the integration of the views. Depending on the particular context, this may result in needs to transfer models among developers and tools, including changing their representations and content, as well as more tight integration of the tools, for example, by accessing one tool from another.
- *Model management*. Proper information management is vital for complex systems, providing services to support versioning, definitions of product variants, handling dependencies between models, consistency management, and traceability among all relevant design entities.
- *Product structure definition*. Closely related to the above point there is a need for information management purposes to define what the product is (e.g., functions, components of different technologies) and how this is related to supporting information such as documentation, workflow, and users. The resulting definition is often called an information model. It is highly desirable that the view models have well-defined connections to the information model.
- *Support for concurrent and distributed engineering*. This type of support includes abilities of simultaneous access and sharing of the provided services across organizations. This imposes derived requirements on the information management and IT-tool infrastructure.
- *Workflow management*. Related to the previous points, it may be desirable to implement and automate some of the workflow among developers, for example, by providing notification of events, such as when the status of a design entity has changed.

10.6.4 Common Arguments against MBD and Pitfalls

Additional arguments that can be useful in the reasoning about MBD adoption include the following:

- *Tools are costly*! This is why the goals, drivers, and the context of MBD have to be considered to determine the suitability of the tools for the given purposes, such that MBD can be introduced when it pays off.
- *Models are difficult to develop, understand, and are not amenable to analysis and synthesis*! This can be true. However, this probably reflects a mismatch in the choice of the MBD technology. The mismatch could be with respect to the context of MBD usage including possibly inadequate training of users, or simply because the chosen MBD technology is immature.
- *Users say "I don't have time to model or to document, I need to work."* This statement can be related to the above mismatches, and/or also illustrate the need of supporting an MBD with a suitable process aligned to work practices. For an MBD approach to be worthwhile it should be possible to develop rational arguments about the needs for modeling and documentation.
- *A model can never capture reality so why bother?* It is well proven from experience that models can be developed for many purposes to capture reality sufficiently well. However, developing models is costly, so the modeling efforts have to be focused and the target and technologies have to be matched.
- *MBD is just about code generation.* There are many other facets to MBDs. For OEMs, transitioning from specification of embedded systems to actually implementing them is a large step and has many implications including maintenance of in-vehicle software and probably larger responsibilities. On the other hand, it gives the OEMs better control of vital vehicle functionality.
- *Can code generators be trusted?* Code generation from models has faced resistance, just like the transition from assembly to high-level programming languages did decades ago. The basic idea is the same—that of providing designers with more powerful tools, thus relieving them of unnecessary detail. Concerns were previously raised whether the compilers would be able to produce efficient and reliable codes. The same concerns are now sometimes still being raised with regard to code generation from models. The gap to implementation should not be underestimated, but this does not mean that compilers and code generators will not do their job. The required qualities including performance of the generated code are also relative to the domain where the code generation is applied. Generated code is already being used in cars and even in aircraft.
- *Overtrust in models and tools.* This is a common pitfall and lesson from reality, and should be taken as a general advice in using MBD. The analysis can only be as good as the model—"the garbage in/garbage out syndrome." Model validation and designer experience are very important aspects

of MBD. It is sound to maintain a certain critical attitude and ongoing checking of plausibility of the models and analysis results.

- *Too detailed models.* This is also an important lesson from reality, and should be taken as a general advice in using MBD. Experience is required to judge the level of detail required—again the efforts have to be focused. In efforts during the 1980s and 1990s in information modeling, the scopes were in many cases set too broad. Modeling everything in too much detail leads to extremely complex models without a clear idea for the concrete usage of the models to support the development.
- *Telecom went into the UML trap 10 years ago—is the automotive industry now heading the same way?* Hopefully no but the answer is not obvious! Many efforts are devoted towards tailoring UML and thus improving its limitations, for example, improving the semantics and integrating it with other models required for automotive embedded systems. However, UML remains a very complex construction and the use of profiles creates a plethora of dialects that may counter the role of UML as a standard. UML also still needs to be proven outside its traditional domains.

10.7 Conclusions

Automotive embedded systems are becoming more and more complex, have high expectations on quality and functionalities, and are safety and cost critical. The involved technologies are also becoming more mature and standardized. There are therefore strong drivers for introducing MBD for automotive embedded systems to increase the development efficiency and effectiveness.

This chapter has provided a framework to facilitate the reasoning and understanding of MBD. While MBD is certainly not a silver bullet, it can assist designers in many ways by providing support for communication, documentation, analysis, and synthesis of designs.

A multitude of technologies for MBD of embedded systems are available to developers. Examples of these technologies include modeling languages, frameworks, model transformations, exchange formats, analysis and synthesis techniques, and tools. Introducing MBD means to introduce rather complex tools in order to handle increasingly complex products. MBD can be adopted for specific design tasks in a limited part of the organization or in a more holistic manner, addressing both technical and management processes. A successful adoption requires a careful consideration of the contextual requirements and MBD technology limitations. A special concern is that of integration and information management, supporting the use of the many types of models and information that are required in developing automotive embedded systems. Moreover, the usage of these tools needs to be supported by suitable processes and guidelines that describe how the technologies should be used.

A challenge is that of connecting the embedded systems world properly with the automotive mechanical engineering. While this, to a large extent, is an organizational and cultural issue, the supporting tools and technologies today are either focused on

MBD for mechanical engineering or on some aspect of embedded systems engineering, creating a need for research on methodology and systems modeling to fill the gaps. Embedded systems are tightly intertwined with their environment, which consequently also needs to be modeled, and analyzed in conjunction with the embedded system. This environment contains other embedded systems, continuous-time systems, as well as humans. Incorporating abstractions of human behaviors is clearly a challenge. The area of HMI needs further attention to support the creativity of developers and in visualizing complex systems; there is a need to involve HMI experts in tool development and research efforts.

The usage of MBD has to go hand-in-hand with the evolution of processes, organizations, and product architectures along the path to improve the capabilities of the automotive industries to deal with embedded systems. The analogy with MBD in mechanical engineering is tempting. With the advent of the CAD, coined already in the early 1960s, mechanical and hardware engineering are ahead of embedded systems tools and methodology. However, MBD in mechanical engineering is still evolving and there seems to be common trends and interests including the desire to capture design rationale, adopt functional abstractions, streamline/improve information management, and improve systems analysis.

The current limitations in MBD technology and methodology are receiving strong attention. Continued and improved interactions between academia and industry, and between academic communities will be required for addressing systems-level MBD challenges. MBD technology will play an increasingly important role for the development of automotive embedded systems.

Acknowledgments

This work has been supported by the Swedish Strategic Research foundation through the SAVE project, by VINNOVA, through the ModComp project and by the European Commission through the ARTIST2 and ATESST projects. We acknowledge inputs to this chapter from the following project colleagues: Niklas Adamsson, Jad El-khoury, Bengt Eriksson, Ola Larses, Ola Redell, Anders Sandberg, Jianlin Shi, and Ulf Sellgren.

References

Internet references to tools, companies, and organizations are listed separately in the end.

1. Architecture and Analysis Description Language (AADL). SAE standard AS5506, issued November 2004.
2. Adamsson, N. Interdisciplinary integration in complex product development—managerial implications of embedding software in manufactured goods. PhD thesis, TRITA—MMK 2007:04. Department of Machine Design, Royal Institute of Technology, Stockholm, Sweden.
3. ARTEMIS Strategic Research Agenda, 1st edn., March 2006, prepared by the members of the ARTEMIS Strategic Research Agenda Working Group.

http://www.artemis-office.org/DotNetNuke/Backgrounddocuments/tabid/58/Default.aspx (accessed October 5).

4. Bouyssounouse, B. and Sifakis, J. (Eds.). *ARTIST Roadmap for Research and Development* (Lecture Notes in Computer Science), 1st edn., Springer, New York, May, 2005.
5. *ATESST project Deliverable D6.1.1—Elicitation of Overall Needs and Requirements on the ADL—Part II*. M. Törngren (Ed.). By the ATESST consortium. EU IST project no. 2004-026976. Available from www.atesst.org.
6. Axelsson, J. Analysis and synthesis of heterogeneous real-time systems. PhD thesis No. 502. Linköping University, Sweden, 1997.
7. Backlund, G. The effects of modeling requirements in early phases of buyer–supplier relations. Licentiate thesis, Linköping Studies in Science and Technology, Thesis No. 812, Department of Mechanical Engineering, Linköping University, Sweden, 2000.
8. Bahill, A.T. and Gissing, B. Re-evaluating systems engineering concepts using systems thinking. *IEEE Transactions on Systems, Man, Cybernetics, Part C*, 28(1):516–527, November 1998.
9. Broy, M. The grand challenge in informatics: Engineering software-intensive systems. *IEEE Computer*, 39(10):72–80, October 2006.
10. Calvez, J.P. *Embedded Real-Time Systems. A Specification and Design Methodology*. John Wiley & Sons, Chichester, England, 1993.
11. Chen, D., Törngren, M., Shi, J., Lönn, H., Gerard, S., Strömberg, M., and Årzén, K.-E. Model-based integration in the development of embedded control systems, a characterization of current research efforts. In: *Proceedings of IEEE Computer Aided Control Systems Design Symposium*, Munich, Germany, October 2006.
12. Clausing, D.P. *Total Quality Development*. ASME Press, New York, 1994.
13. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J., *Documenting Software Architectures: Views and Beyond*. Addison Wesley, Reading, MA, 2002.
14. Cooling, J. *Software Engineering for Real-Time Systems*. Pearson Education Limited, Harlow, England, 2003.
15. Courtois, P.J. and Parnas, D.L. Documentation for safety critical software. In: *Proceedings of the 15th IEEE International Conference on Software Engineering*, Baltimore, MD, May 1993, pp. 315–323.
16. Crnkovic, I., Asklund, U., and Persson, D.A. *Implementing and Integrating Product Data Management and Software Configuration Management*. Artech House Publishers, Norwood, MA, 2003.
17. Cuenot, P., Chen, D., Gérard, S., Lönn, H., Reiser, M.-O., Servat, D., Tavakoli Kolagari, R., Törngren, M., and Weber, M. Improving dependability by using an architecture description language. In: Lemos, R., Gacek, C., and Romanovsky, A. (Eds.), *Architecting Dependable Systems IV*. Springer series: Lecture Notes in Computer Science, Vol. 4615, Berlin, Germany, 2007.
18. Czarnecki, K. and Helsen, S. Classification of model transformation approaches. In: *Proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, CA, 2003.
19. Damm, W. and Metzner, A. A design methodology for distributed real-time automotive applications. In: *Next Generation Design and Verification Methodologies*

for Distributed Embedded Control Systems, Springer LNCS, Berlin, Germany, 2007, pp. 157–174.

20. Daniels, J. Modeling with a sense of purpose. *IEEE Software*, 19(1):8–10, January/February 2002.
21. El-khoury, J., Chen, D., and Törngren, M. 2003. A survey of modeling approaches for embedded computer control systems, Technical Report, Royal Institute of Technology, Stockholm. TRITA-MMK 2003:36, Sweden, 2003.
22. El-khoury, J. A model management and integration platform for mechatronics product development, PhD thesis, Department of Machine Design, KTH, 2006:03, May 2003.
23. El-khoury, J., Redell, O., and Törngren, M. A model and tool integration platform for multidisciplinary development. In: *Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications*, Porto, Portugal, 2005.
24. Engelhardt, F. Improving systems by combining axiomatic design, quality control tools and designed experiments. *Research in Engineering Design*, 12:204–219, 2000.
25. Freude, R. and Königs, A. Tool integration with consistency relations and their visualization. In: *Ninth European Software Engineering Conference and 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Helsinki, Finland, 2003.
26. Garetti, M. and Terzi, S. Organisational change and knowledge management in PLM implementation. *International Journal of Product Lifecycle Management*, 1:43–51, 2005.
27. Gero, J.S. 1990. Design prototypes: A knowledge representation scheme for design. *AI Magazine*, 11(4):26–36, 1990.
28. Gomaa, H. *Software Design Methods for Concurrent and Real-time Systems*, Addison-Wesley, Boston, MA, 1993.
29. Goševa-Popstojanova, K. and Trivedi, K.S. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2–3):179–204, July 2001.
30. Hansen, M., Nohria, N., and Tierney, T. 1999. What's your strategy for managing knowledge? *Harvard Business Review*, 77(2):106–116, 1999.
31. Heinecke, H., Schnelle, K.-P., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Mat/e, J.-L., Nishikawa, K., and Scharnhorst, T. Automotive open system architecture an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. In: *Proceedings of Convergence 2004*, Detroit, MI, October 2004.
32. ANSI/IEEE Standard 1471-2000, Recommended practice for architectural description of software-intensive systems, September 2000.
33. Jezequel, J.M. and Meyer, B. Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, January 1997.
34. Karsai, G., Lang, A., and Neema, S. Design patterns for open tool integration. *Software and Systems Modeling*, 4(2):157–170, May 2005.
35. Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T. Model integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.
36. Kleppe, A., Warmer, J., and Bast, W. *MDA Explained. The Model Driven Architecture: Practice and Promises*. Addison-Wesley, Boston, MA.
37. Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, MA, 1997.

38. Knippel, E. and Schulz, A. Lessons learned from implementing configuration management within electrical/electronic development of an automotive OEM. In: *Proceedings of the 14th International Symposium of the International Council on Systems Engineering*, Toulouse, France, June 20–24, 2004.
39. Kruchten, P. Casting software design in the function-behavior-structure framework. *IEEE Software*, 22(2):52–58, 2005.
40. Larses, O., Sjöstedt, C.-J., Törngren, M., and Redell, O. Experiences from model supported configuration management and production of automotive embedded software. In: *Proceedings of the SAE World Congress, In-Vehicle Software Session*, Detroit, MI, 2007.
41. Larses, O. and Adamsson, N. Drivers for model based development. In: *Proceedings of the Eighth International Design Conference on Design*, Dubrovnik, Croatia, 2004.
42. Larses, O. Architecting and modeling automotive embedded systems. PhD thesis, Department of Machine Design, Royal Institute of Technology, Stockholm. TRITA-MMK 2005:31, November 2005.
43. Lee, E.A. and Sangiovanni-Vincentelli, A. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
44. Leveson, N.G. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Company, Reading, MA, 1995.
45. Lindahl, M. Engineering designers' requirements on design for environment methods and tools. Doctoral thesis in machine design, Royal Institute of Technology, Stockholm, Sweden, 2005.
46. Lohmar, W. The virtual development process—A reality at SEAT. In: *Proceedings of the FISITA 2004 30th World Automotive Congress*, Reference F2004F450, Barcelona, Spain, May 23–27, 2004.
47. Lowe, A., Mcmahon, C., and Culley, S. Characterising the requirements of engineering information systems. *International Journal of Information Management*, 24:401–422, 2004.
48. Maier, M. and Rechtin, E. *The Art of Systems Architecting.* CRC Press, Boca Raton, FL, 2002.
49. Malvius, D. Information management for complex product development. Licentiate thesis, TRITA-MMK 2007:09, Department of Machine Design, KTH, August 2007.
50. Malvius, D., Redell, O., and Ritzén, S. Introducing structured information handling in automotive EE development. In: *Proceedings of the 16th International Symposium of the International Council on Systems Engineering INCOSE*, Orlando, FL, July 2006.
51. McDermid, J.A. Assurance in high integrity software. In: *High Integrity Software*, Sennet, C.T. (Ed.). Pitman, London, United Kingdom, 1989.
52. MDA Guide Version 1.0.1. Document Number: omg/2003-06-01. Date: 12 June 2003.
53. Medvidovic, N. and Taylor Richard, N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
54. Hailpern, B. and Tarr, P. Model-driven development: The good, the bad and the ugly. Special issue on model-driven software development. *IBM Systems Journal*, 45(3):451–462, 2006.

55. Nambisan, S. and Wilemon, D. Software development and new product development: Potentials for cross-domain knowledge sharing. *IEEE Transactions on Engineering Management*, 47(2):211–220, 2000.
56. Nonaka, I. and Takeuchi, H. 1995. *The Knowledge-Creating Company*. Oxford University Press, New York, 1995.
57. Norell, M. Stödmetoder och samverkan i produktutveckling (Swedish). Doctoral thesis (report TRITA-MAE-1992:7), Department of Machine Elements, Royal Institute of Technology, Stockholm, Sweden, 1992.
58. Oliver, D.W., Kelliher, T.P., and Keegan, Jr., J.G., *Engineering Complex Systems with Models and Objects.* McGraw-Hill, New York 1996.
59. Ottersten, I. and Balic, M., Effektstyrning av IT, Liber AB, 2004.
60. Papadopoulos, Y., McDermid, J., Sasse, R., and Heiner, G. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering* & *System Safety* 71:229–247, 2001.
61. Pahl, G. and Beitz, W., *Engineering Design—A Systematic Approach.* Springer-Verlag, New York, 1996.
62. Prasad, B. *Concurrent Engineering Fundamentals, Volume I: Integrated Product and Process Organization*. PTR Prentice Hall, New Jersey, 1996.
63. Ranville, S. Case study of commercially available tools that apply formal methods to a Matlab/Simulink/Stateflow model. SAE World Congress, Detroit, MI, March 8–11, 2004.
64. Redell, O., El-khoury, J., and Törngren, M. The AIDA toolset for design and implementation analysis of distributed real-time control systems. *Microprocessors and Microsystems*, 28(4):163–182, May 2004.
65. Reichmann, C., Kuhl, M., Graf, P., and Muller-Glaser, K.D. GeneralStore—A CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In: *11th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*—ECBS 2004, Brno, Czech Republic, May 24–26, 2004.
66. Rushby, J. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-1. Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995.
67. SAE ARP-4761: *Aerospace Recommended Practice: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 12th edn., SAE, 400 Commonwealth Drive Warrendale, PA, 1996.
68. Sander, I. and Jantsch, A. System modeling and transformational design refinement in Forsyde. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.
69. Sangiovanni-Vincentelli, A. and Quo, V. SLD? Reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.
70. Sangiovanni-Vincentelli, A. Automotive electronics: Trends and challenges. In: *Convergence 2000*, Detroit, MI, October 2000.
71. Schaetz, B., Pretschner, A., Huber, F., and Philipps, J. Model-based development. Technical report—TUM-I0204. Institut fur Informatik, Technical University of Munich, TUM-INFO-05-I0204–0/1.-FI, May 2002.
72. Schmidt, D.C. Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.

73. Selic, B., Gullekson, G., and Ward, P. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
74. Sellgren, U. Simulations in product realization—A methodology state of the art report. Technical report, Department of Machine Design, Royal Inst. of Technology, 2003.
75. Sharman, D.M. and Yassine, A.A. Characterizing complex product architectures. *Journal of Systems Engineering*, 7(1):35–60, March 2004.
76. Shi, J., Törngren, M., Servat, D., Sjöstedt, C.-J., Chen, D., and Lönn, H. Combined usage of UML and Simulink in the design of embedded systems: investigating scenarios and structural and behavioral mapping. In: *Proceedings of the OMER4 Workshop: Object-Oriented Modeling of Embedded Real-Time Systems*, Paderborn, October 2007.
77. Siegers, R. The ABCs of AFs: understanding architecture frameworks. In: *Proceedings of INCOSE International Symposium 2005*, Rochester, NY, July 10–15, 2005.
78. Stevens, R., Brook, P., Jackson, K., and Arnold, S. *Systems Engineering—Coping with Complexity*. Pearson Education, Harlow, England, 1998.
79. Sztipanovitz, J. and Karsai, G. Embedded software: Challenges and opportunities. In: *Proceedings of EMSOFT 2001*, LNCS 2211, Springer-Verlag, Berlin Heidelberg, Germany, 2001, pp. 403–415.
80. Törngren, M. Fundamentals of implementing real-time control applications in distributed computer systems. *Journal of Real-Time Systems*, 14:219–250, 1998.
81. Törngren, M., Chen, D., and Crnkovic, I. Component based vs. model-based development: A comparison in the context of vehicular embedded systems. In: *Proceedings of 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, Porto, Portugal, 30 August–3 September 2005.
82. Törngren, M. and Larses, O. Maturity of model driven engineering for embedded control systems from a mechatronic perspective. In: *Model Driven Engineering for Distributed Real-Time Embedded Systems*, Gérard, S., Babau, J.-P., and Champeau, J. (Eds.), ISTE, London, September 2005.
83. Törngren, M., Henriksson, D., Redell, O., Kirsch, C., El-Khoury, J., Simon, D., Sorel, Y., Zdenek, H., and Årzén K.-E. Co-design of control systems and their real-time implementation—A tool survey. Technical Report, Department of Machine Design, KTH, August 2006. TRITA-MMK 2006:11.
84. Westfechtel, B. and Conradi, R. Software configuration management and engineering data management: Differences and similarities. In: *Proceedings of the Eighth International Workshop on System Configuration Management*, Springer-Verlag, 1998, pp. 95–106.
85. VDI, *Design Methodology for Mechatronic Systems. VDI 2206–Verein Deutscher Ingenieure*. Beuth Verlag Fmbh, 10772 Berlin, Germany, 2004.
86. Vesely, W.E. *Fault Tree Handbook*. U.S. Nuclear Regulatory Committee Report NUREG-0492, US NRC, Washington DC, 1981.
87. Wild, D., Fleischmann, A., Hartmann, J., Pfaller, C., Rappl, M., and Rittmann, S. An architecture-centric approach towards the construction of dependable automotive software. In: *SAE 2006 World Congress*, Detroit, MI, April 2006, Session: Systems Engineering. SAE paper no. 2006-01-1222.
88. Zeigler, B.P., Praehofer, H., and Kim, T.G. *Theory of Modeling and Simulation*, 2nd edn., Academic Press, New York, 2000.

Internet References—October 2007

89. AADL, www.aadl.info
90. ARTISAN studio, http://www.artisansw.com/products/
91. ASCET, http://www.etas.com/en/products/ascet_software_products.php
92. ASAM, http://www.asam.net/
93. ATESST, Advancing traffic efficiency and safety through software technology: www.atesst.org
94. Autosar, http://www.autosar.org/
95. COM, Component Object Model Technologies: http://www.microsoft.com/com/default.mspx
96. CORBA, Object Management Group. Catalog of OMG Specifications: http://www.omg.org/technology/documents/spec_catalog.htm
97. DOORS, http://www.telelogic.com/corp/Products/doors/doors/index.cfm
98. dSPACE, www.dspace.de and SystemDesk: http://www.dspace.com/ww/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm?nv=lb
99. Eclipse, The Eclipse Project, http://www.eclipse.org/
100. ISO, http://www.iso.org/iso/home.htm
101. MARTE, ProMarte consortium. Joint UML Profile for MARTE Initial Submission, realtime/05–11–01, November 2005, available at: http://www.omg.org/cgi-bin/doc?realtime/05-11-01.
102. Matlab and Simulink, Mathworks: http://www.mathworks.com/
103. Metropolis, http://embedded.eecs.berkeley.edu/metropolis/forum/2.html
104. Milan, Model-based integrated simulation: http://milan.usc.edu/
105. Misra guidelines, http://www.misra.org.uk/
106. Modelica and the Modelica Association, http://www.modelica.org/
107. MOF, http://www.omg.org/mof
108. Ptolemy, http://ptolemy.eecs.berkeley.edu/
109. RIF, The Requirements Interchange Format: http://www.automotive-his.de/rif/
110. STEP, Overview of STEP (ISO 10303) Product Data Representation and Exchange: http://www.tc184-sc4.org/SC4_Open/SC4%20Legacy%20Products%20(2001–08)/STEP_(10303)/
111. SysML, http://www.sysml.org/
112. SystemC, http://www.systemc.org
113. TT-tech, http://www.ttautomotive.com/
114. The Unified Modeling Language, http://www.uml.org/
115. Vector, http://www.vector-cantech.com/va_davinci_us,,4165.html
116. Volcano, http://www.mentor.com/solutions/automotive/automotive_networking.cfm
117. W3C, http://www.w3.org/
118. XMI specification, http://www.omg.org/technology/documents/formal/xmi.htm

IV

Verification, Testing, and Timing Analysis

11

Testing Automotive Control Software

Mirko Conrad
The MathWorks, Inc.

Ines Fey
Safety and Modeling Consultants

11.1 Introduction

11.1.1 Dynamic Testing

“Dynamic testing” (in short, testing) is one of the most significant analytical quality assurance techniques for software, as it is the most elementary and certainly the most frequently used form of quality assurance [Lig92]. In practice, it is also the only method that allows to take the actual development and operating environment of a software system (e.g., code generator, compiler, linker, operating system, target hardware) adequately into consideration. Furthermore, the dynamic properties of the system (e.g., run-time behavior, computational accuracy of the target system) can be checked [Gri95]. Dynamic testing is the most important and common method used to assure the quality of automotive controls and is an essential part of both software and system development. The artifacts to be tested are usually named "test objects"* or "test items." Based on Refs. [Gri88,ISO 15497,IEEE 610.12,TAV96], dynamic testing can be defined as the execution of a "test object" on a computer with selected (sequences of) test inputs in a defined environment for the purpose of checking whether the test object in such cases behaves as expected.

As opposed to other analytical quality assurance techniques, dynamic testing, according to Liggesmeyer [Lig90], is characterized as follows:

- Test object is dynamically executed with (sequences of) test inputs.
- Test object is being tested in its real environment and/or a simulation of such an environment.
- Testing is a sampling procedure, the correctness of the test object with regard to expected behavior cannot be verified in the mathematical sense.

Due to the constraints and characteristics of embedded systems, testing of such systems and their embedded software differs significantly from testing administrative or scientific–technical software. Practicable test approaches should therefore be tailored to the specifics of embedded systems and need to address time and sequencing and, in particular, allow for the design of time-dependent test scenarios. Existing test approaches from other domains are only of limited use in this domain.

11.1.2 Current Practice

A core element of quality assurance in automotive-embedded systems [SZ05] is "in-vehicle testing" or "road testing." Road testing, which actually serves to calibrate control parameters and to validate the entire vehicle, is, however, often misused to find software-related specification, design and implementation errors [Sim97]. Road tests swallow up a considerable amount of time and effort. It is not unusual for the testing of an individual control unit to require distances of several million kilometers to be traveled. Furthermore, road tests often occur at a critical stage (in terms of time) of controls development, as they normally take place at the end of the systems development. At that time, the average costs for error correction are already very high. Due to the considerable time and cost pressure, usually a compromise is made between early market introduction and the achieved testing depth. As a result, development and implementation errors in embedded software are discovered too late in the development process or even by the customers themselves [Sim97].

* We use the term test object rather than "system under test" (SUT) to emphasize the variety of possible test objects that includes systems as well as parts of a system, and precursory stages (models) of a system.

Compared to road testing, the application of methods and procedures for "systematic testing of software" during its development enables earlier and more efficient detection of software-related errors. However, this requires a practically oriented adaptation of the general testing techniques and approaches to the specifics of the application domain. To ensure high effectiveness, the tests should be carried out during the development process rather than just at the end of the development.

However, it can often be observed that testing of today's automotive control software follows a "gut-feel approach," leading to test gaps and test redundancies.

Since exhaustive testing is impossible in practice, dynamic testing is always a sampling procedure. A subset of test scenarios, which is as small as possible and which is able to reveal as many errors in the test object as possible, needs to be selected from the (infinite) set of all possible test scenarios. The selection of adequate sample elements (test scenarios) decides on the extent and quality of the entire test (Figure 11.1).

In Figure 11.1, the input domain of the test object is depicted as a black-framed rectangle, the areas marked gray (e.g., ①, ②) illustrate erroneous subdomains. Those parts of the input domain that are covered by test scenarios are presented as circles. If a test scenario lies within a gray area (e.g., ③) this means that it is an error-revealing test scenario, otherwise (e.g., ④) it is nonerror revealing.

Because of the exceptional significance of the selection process, there is a need for "systematic test design" techniques which support the tester when designing adequate test scenarios. The application of these test techniques systematizes the selection of test scenarios and makes it both comprehensible and reproducible [Sim97].

When testing automotive control software, an "ad hoc selection of test scenarios" is typical, that is, the test scenarios are determined without (explicitly) defined procedures. The actual selection depends on the experience and expertise of the tester and can only be reproduced with difficulty, if at all. An ad hoc selection of test scenarios leads, on the one hand, to test scenarios that are more or less redundant (upper right area in Figure 11.1) and, on the other hand, to test gaps (lower left area in Figure 11.1). Moreover, "one-factor-at-a-time testing" is typical, that is, only variations in single influencing variables take place (test scenarios arranged like beads on a string

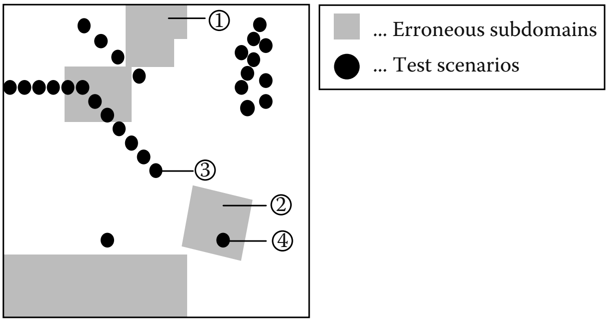


FIGURE 11.1 Input domain, errors, and test scenarios.

in Figure 11.1). As a rule, it is impossible to make statements about the quality or completeness of the ad hoc test scenarios. With regard to the number of test scenarios, their level of error detection is often inadequate.

Those software test design techniques that are methodically well-founded and that exist in the area of general software engineering are often tailored toward administrative or scientific software. They are not sufficiently adapted to the specifics of embedded automotive control systems and can therefore not be transferred just like that. The temporal aspects of the test scenarios, in particular, are not suitably taken into account.

11.1.3 Structuring the Testing Process

Testing large embedded automotive systems is a complex endeavor, involving many specialists carrying out different test levels and test activities at various stages of the project.

In order to structure the test process and to facilitate the test complex systems, different "test phases" have been established in test theory that normally resemble the chosen integration strategy. In theory, test phases include unit testing, integration testing, and system testing. In practice, the situation is more complex: Throughout the project numerous artifacts are tested in different environments by various testers and test teams at various points of time in the project. Such organizational aspects are the reason for introducing the notion of "test levels." A test level is a distinct group of activities that is planned and managed as an entity [BN03]. So, the test levels occurring in a particular project are not only defined by test theory, but also by software engineering practice. The test levels to be applied may also differ between code-based (Section 11.3.1.1) and model-based development (Section 11.3.2.1).

Usually, each test level can be carried out separately but following a specified sequence of "test activities" such as "test scenario design," test execution, and test evaluation (Section 11.2.1). This means the whole bunch of test activities has to be performed on a per test level basis. In order to facilitate efficient testing it is recommended to define distinctive objectives for each test level and to reuse artifacts and data across test levels as far as feasible.

11.1.4 Model- versus Code-Based Testing

In order to facilitate an efficient testing process it is absolutely essential to adapt the testing process to the needs of the overall development process. Hereby, the "software development paradigm" applied during the control software development substantially influences the testing process. For the remainder of the chapter we distinguish the following two popular development paradigms:

"Code-based development" is used here to denote the classic software development paradigm in which a clear demarcation of the constructive development phases is recognizable from specification to design to implementation, and in which the applied procedures, particularly for coding, are manual or text based and not model based.

By contrast, “model-based development” (Chapter 10) focuses on a model central to all development phases, which is executable and typically graphic. This model is usually created at an early stage of the development process and includes three parts that can be simulated in combination: the controller model (or functional model), a model of the controlled system (vehicle model), and the model of the system’s environment (environmental model). The model of the controlled system and the environmental model are gradually replaced by the real system and its real environment as the development process continues. The functional model evolves over time and finally serves as the basis for implementing the embedded software on the electronic control unit through manual or automatic coding. Vehicle models and environmental models are referred to as plant models in control engineering. Modeling notations commonly employed are block diagrams and extended state machines provided by commercial modeling and simulation packages as the Simulink product family [SL,SF].

A substantial difference between these development paradigms, with respect to testing, arises from the different test objects that must be taken into consideration in the testing process. In code-based development, typical artifacts that can be tested are the control software on a host platform, the control software on a target platform, and the integrated embedded system. In model-based development different types of models (e.g., design model, implementation model) can be tested additionally.

Testing within a code-based development process is referred to as “code-based testing,” whereas testing within model-based development is termed “model-based testing.”

The following sections intend to give an overview of best practices in testing embedded automotive systems. Since application development takes a prominent place in automotive systems development we focus here on testing the application part of the system [SZ05] (see Chapter 3). Other aspects like network features are of great interest as well, but they are not detailed in this chapter (see Chapter 12). Section 11.2 discusses core test activities and testing techniques to support these. Section 11.3 shows how testing can be integrated into model- and code-based control software development processes. Section 11.4 covers test planning.

11.2 Test Activities and Testing Techniques

11.2.1 Test Activities

The term “testing” not only describes the activity of testing itself, but also the design and evaluation of tests [TAV96]. Subdividing testing into individual “test activities” forms the basis for a systematic test and promotes a structured procedure. A better documentation and traceability entail [Weg01]. The subdivisions given in the literature vary (see Refs. [Gri95,BN03]), but it is often possible to distinguish “core activities” and “supporting activities.” For carrying out the different test activities in a methodological way, numerous “testing techniques” have been developed.

This section will start off by presenting the test activities test scenario design, test execution, and test evaluation. It will then introduce exemplary testing techniques for these three test activities. The test activities and testing techniques presented are, for the time being, independent of a specific test level.

11.2.1.1 Test Scenario Design

Among the test activities, the design of suitable, that is, error-sensitive, test scenarios (test scenario design) is the most crucial activity for a trustworthy test of automotive control software.

A "test scenario" (Figure 11.2) is a finite structure of test inputs and expected outputs: a pair of input and output in the case of deterministic transformative systems, a sequence of inputs and outputs in the case of deterministic reactive systems, and a tree or a graph in the case of nondeterministic reactive systems. A "test suite" is a finite set of test scenarios [UPL06]. If test scenarios comprise a notion of time, then they are termed "timed test scenarios"; if they do not comprise a notion of time they are termed "untimed test scenarios."

Since an exhaustive test is practically impossible even for smaller systems due to combinatorial reasons, it is necessary to choose a subset as small as possible from the set of potential test scenarios, which ensures that the test object is tested as comprehensively as possible [Mye79]. This selection process causes the sampling character of dynamic testing [Lig92]. The selection of suitable samples (i.e., test scenarios) determines the extent and quality of the test.

Due to the outstanding significance of the design of test scenarios, numerous techniques have been developed in the past that support the tester in the selection of suitable test scenarios [Weg01]. The application of these test design techniques systemizes the selection of test scenarios and makes them traceable and reproducible [Sim97].

In the following, "test design technique" stands for a standardized, but not necessarily formal technique of selecting and/or designing test scenarios based on a certain source of information, termed "test basis." Thereby, a test design technique includes a suitable notation for the description of the selected test scenarios. Possible test bases could, among other things, be the requirements specification, a design model, or the program code created from it.

The multitude of imaginable criteria for the selection of test scenarios is also responsible for the high number of existing test design techniques. For the classification of test design techniques it is possible to draw on different criteria [Mye79, Lig92,Gri95,Bal98,Bel98,Con01]. Typically however, test design techniques are distinguished by whether or not the structure of the test object is taken into account

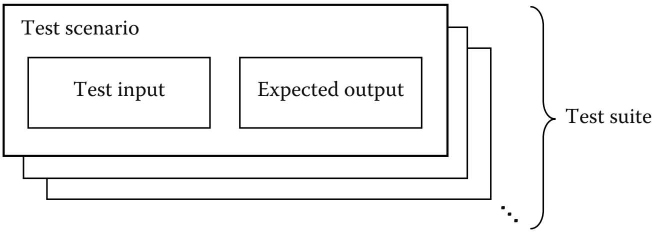


FIGURE 11.2 Elements of a test scenario.

during the selection of the test scenarios. A finer distinction is possible if, in addition, there is a classification according to the test basis.

Techniques, in which the internal structure of the object to be tested is irrelevant for the determination of test scenarios, are called "functional" or "black-box test design techniques," otherwise "structural" or "white-box test design techniques." Procedures in which partial knowledge of the internal structure is required are called "gray-box techniques."

The objective of the functional techniques is to test the specified function of the test object as comprehensively as possible. The objective of the structural techniques is to cover certain structural elements of the object to be tested as exhaustively as possible. Structural elements to be covered can be defined by the control flow (control flow-related test design) or the data flow (data flow-related test design) of the test object.

However, the distinction into functional and structural techniques—in particular in the case of black-box techniques—does not provide any statements about the source from which the test scenarios are derived. The only requirement is that the derivation of the test scenarios in the black-box test does not result from the program structure. The test basis offers a further classification criterion for the test design techniques.

If the test scenarios are derived from the usually textual, functional specification (from the test object, only the interfaces might be taken into account), this is called "specification-oriented test design." If a model of the test object serves as the basis, this is called "model-oriented test design." Finally, if the source code is used as the test basis, this is called "code-oriented test design."

Description of different general test design techniques can be found, among others, in Refs. [Mye79,Lig90,Bal98,BN03]. Due to the characteristics of embedded automotive systems, applicable test design techniques for control software differ from techniques developed for administrative or scientific–technical software. Practical test design techniques should therefore be tailored to these specifics and need to address time and sequencing. In particular, they must allow for the description of timed test scenarios. Existing test approaches from other domains are only of limited use for this.

As the use of a single test design technique is usually not sufficient for a thorough test of automotive control software, techniques that complement each other must suitably be combined. Thus, in order to determine test scenarios, it is typical to consult different sources of information (textual specifications, design models, if necessary earlier models or separate test models) and to combine functional and structural techniques. Combinations of different test design techniques are termed "test approaches." Usually, they include not only test design techniques, but also techniques for other test activities, for example, test evaluation.

11.2.1.2 Test Execution

Within the scope of "test execution" a test object is stimulated with the test inputs established in the test scenario. The system reaction resulting from this is recorded.

Since the industrial development of automotive software does not take place on an ad hoc basis but in different phases, it is not only the end product that is tested in a test process accompanying the development, but also its different precursory stages (model evolution stages). In the context of a model-based development process emerges, for example, at first a model of the automotive control software (possibly in varying abstract forms of representation: design model, implementation model). This will later be translated into program code (possibly in several stages: floating/fixed point code), which is then loaded onto a corresponding target hardware (possibly also in several stages: prototype hardware, production electronic control unit [ECU]). Each of these development artifacts (representation forms) can be tested. The test of the executable model of the software is called "model test" or "model level test." The test of the thus generated software on the development computer shall be termed "software test" (in the strict sense) or "software level test," while the test of the software on a target system is called the "test of the embedded system" or "system level test." In the scope of traditional, code-based development the model tests do not apply.

If more than one representation form of a test object is being tested, the results of earlier tests can be used as a "test reference," that is, as expected output for the test of another form of representation. Such a "comparing test" is called "back-to-back test." Another variant of comparing tests is "regression tests," in which two different versions of one and the same representation form of the test object are tested against each other.

For test execution we usually need a "test environment," which ensures the feed-in of the test object with the test inputs and the recording of the system reaction. The latter has to be saved persistently, at least until the test evaluation. Depending on the type of the test object and the execution environment used it is possible for the test environment to take on different forms; "stubs" and "drivers" in the case of code-based software level tests, "test harnesses" in the form of models with model level tests.

Depending on whether or not the plant or a plant model is included in the test execution, one speaks of "closed-loop" or "open-loop tests." In the case of a closed-loop test execution the relevant parts of the plant or the plant model have to be integrated into the test environment. In the case of closed-loop control components it is possible to thus feed back, for example, actual values.

In order to measure the structural test coverage during the test execution, it is necessary to "instrument the test object" (model or source code).

Testing techniques that support test execution are designated as "test execution techniques."

11.2.1.3 Test Evaluation

In the scope of the test evaluation the system reaction is compared to the expected behavior by taking into consideration defined "acceptance criteria." The comparison result is then together with information about the test execution condensed into a "test verdict" (Figure 11.3).

If a "test oracle" by means of a model for the behavior of the test object exists, one can also evaluate the actual system reaction with the help of the oracle model. In this case, test evaluation is possible without explicit a priori determination of the expected outputs during test design.

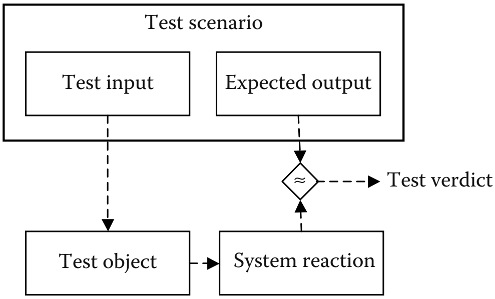


FIGURE 11.3 Test execution and evaluation.

The comparison between system reaction and expected behavior can take place in different ways: The easiest form of the test evaluation is the "manual visual evaluation" (evaluation by "careful watching"). For this, the recorded system reactions are processed in a suitable form (e.g., through plots of the system reaction) and then are submitted to the tester for the visual inspection. A special case of the manual visual test evaluation is to use a photo realistic or schematic "animated system reaction" [CH98,RCK+00]. For this, variables or states of the test object are coupled with graphic and numeric elements of a vehicle animation or elements of a visualization panel. This way, it is possible to graphically depict, for example, vehicle variables such as velocity, angle of rotation, etc. In addition, vectorial variables can be visualized by means of fading-in the respective vector arrows.

"Manual, nonvisual evaluations" are applied whenever we test directly in the vehicle and if the nonvisual sensory perceptions of human beings are needed for evaluating the system reaction qualitatively. This type of test evaluation is used, in particular, during the evaluation of comfort aspects, for example, during the evaluation of the acceleration behavior of an adaptive cruise control system. In colloquial speech, the manual nonvisual evaluation is also called *ass-o-meter testing.*

Manual evaluations are applied whenever neither automatically usable expected outputs nor reference outputs are available or the complexity of the system reaction to be analyzed makes an automated evaluation impossible. The disadvantages are the error-proneness and the high time effort for the evaluation, which makes a significant difference in particular with large test suites.

If the correctness of the system reaction can be determined by calculating Boolean predicates and if these are checked by "watchdogs," which are integrated into the test execution environment, test evaluation can be reduced to calculating logical predicates that use the output signals of the watchdogs. This means, that an automated test evaluation takes place via the evaluation of the watchdog outputs. This way, it is possible to monitor, for example, controller characteristics or dynamic signal boundaries. Examples can be found in Refs. [CH98,Rau02].

While the comparison of system reaction and expected behavior in the case of untimed test scenarios is relatively simple, we will have to, in the case of timed test scenarios, compare system reactions in the form of output time series with expected behavior or reference values in the form of reference time series. The problem of the test evaluation can then be reduced to a *robust signal comparison* between the signals (time series) of the system reaction and the corresponding signals of the reference data or the expected output data.

Furthermore, in the context of the test evaluation it is necessary to decide whether the behavior of the test object can be accepted as correct or has to be declined as incorrect. In this, the comparison results are assigned to one of the three classes "pass," "fail," or "inconclusive" (*traffic light assessment*). Combined with information about the status of the test execution (classes "error," "undefined") the *test verdict* arises. Depending on the type of the test execution and evaluation it is possible to execute the formation of verdicts either manually or automatically. In practice, there is often a combination of automated pre-evaluation with manual approval so that the responsibility for the (non)release of a test remains with a human being.

In addition, it is necessary to check during test evaluation, whether the established test objectives and test criteria are fulfilled to the desired degree by the test. In the case of functional tests, it is possible to determine and analyze, for example, the reached "requirements coverage" or "data range coverage," or, in the case of structural tests, the reached "structural model or code coverage." If the reached degree of coverage is insufficient, it is necessary to analyze the reasons and, usually, to complement further test scenarios.

Testing techniques that support test evaluation are called "test evaluation techniques" in the following.

11.2.2 Exemplary Test Design Techniques for Automotive Control Software

In the following, one functional and one structural test design technique tailored to the particularities of the object context will be presented.

11.2.2.1 Functional Test Design: The Classification-Tree Method for Embedded Systems CTM$_{\text{EMB}}$

The classification-tree method for embedded systems (CTM$_{\text{EMB}}$) (formerly known as CTM/ES) [CDF+99,Con04a,Con04b,CK06], an extension of the classification-tree method [GG93], provides a functional test design technique that allows systematic test design for automotive control software based on a functional specification and an interface description of the test object as well as a comprehensive graphical description of timed test scenarios by means of abstracted time series that are defined stepwise for each input.

Based on an interface description of the test object, the input domain of the test object is split up according to different aspects usually matching the various inputs. The different partitions, called "classifications," are subdivided into (test input data) equivalence *classes*. Finally, different combinations of input data classes are selected and arranged into "test sequences."

For illustration purposes a simplified feature which analyzes the pedal positions in a car is used. This functionality can be employed as a preprocessing component

TABLE 11.1 Software Requirements Specification of PedInt (Excerpt)

ID	Description
SR-PI-01	Recognition of pedal activation If the accelerator or brake pedal is depressed more than a certain threshold value, this is indicated with a pedal-specific binary signal
SR-PI-01.1	Recognition of brake pedal activation If the brake pedal is depressed more than a threshold value ped_min, the BrakePedal flag should be set to the value 1, otherwise to 0
SR-PI-01.2	Hysteretic behavior of brake pedal activation No hysteresis is to be expected during brake pedal activation recognition
SR-PI-01.3	Recognition of accelerator pedal activation If the accelerator pedal is depressed more than a threshold value ped_min, the AccPedal flag should be set to the value 1, otherwise to 0
SR-PI-01.4	Hysteretic behavior of accelerator pedal activation No hysteresis is to be expected during accelerator pedal activation recognition
SR-PI-02	Interpretation of pedal positions Normalized pedal positions for the accelerator and brake pedal should be interpreted as desired torques. This should take both comfort and consumption aspects into account
SR-PI-02.1	Interpretation of brake pedal position [...]
SR-PI-02.2	Interpretation of accelerator pedal position [...]

for various chassis control systems. The pedal interpretation (PedInt) component interprets the current, normalized positions of accelerator and brake pedal (phi_Acc, phi_Brake) by using the actual vehicle speed (v_act) as desired torques for drivetrain and brake (T_des_Drive, T_des_Brake) (see Ref. [Höt97] for details). Furthermore, two pedal position flags (AccPedal, BrakePedal) are calculated that indicate whether or not the pedals are considered to be depressed.

Table 11.1 summarizes the software requirements for the PedInt component. For the following considerations, those subfunctions of PedInt should be tested that are responsible for the calculation of the two flags, that is, all requirements with ID's SR-PI-01.x.

As a first step, the interface that is relevant for the test has to be determined.

Given a test object with n inputs and m outputs. Then, the input interface I of the test object is represented via the input variables $i_i(i=1,\ldots,n)$, the output interface O via the output variables $o_j(j=1,\ldots,m)$. In order to test the test object, each input variable of the effective test interface has to be stimulated with a timed signal, that is, a time series of data. The output variables have to be recorded. Thus, the test interface $I \cup O$ determines the "signature of the test scenarios" belonging to the given test object

Figure 11.4 shows a Simulink [SL] model of the PedInt component that serves as a test object in the following example; Table 11.2 describes its test interface $I \cup O$.

Thus, the signature for the component test of PedInt is as follows:

$$\begin{array}{lll} I: & \text{phi\_Acc}^*, \text{phi\_Brake}^*: & \text{Time} \to R\big|_{[0,100]} \\ & v\text{\_act}^*: & \text{Time} \to R\big|_{[-10,100]} \end{array}$$

$$\begin{array}{lll} O: & T\text{\_des\_Drive}^*, T\text{\_des\_Brake}^*: & \text{Time} \to R \\ & \text{AccPedal}^*, \text{BrakePedal}^*: & \text{Time} \to \{0,1\} \end{array}$$

The input domain of the test object, which can be regarded as a (n-dimensional) hypercube formed by the admissible ranges of its (n) different input signals, is disjointedly and completely partitioned into input equivalence classes that are suitable

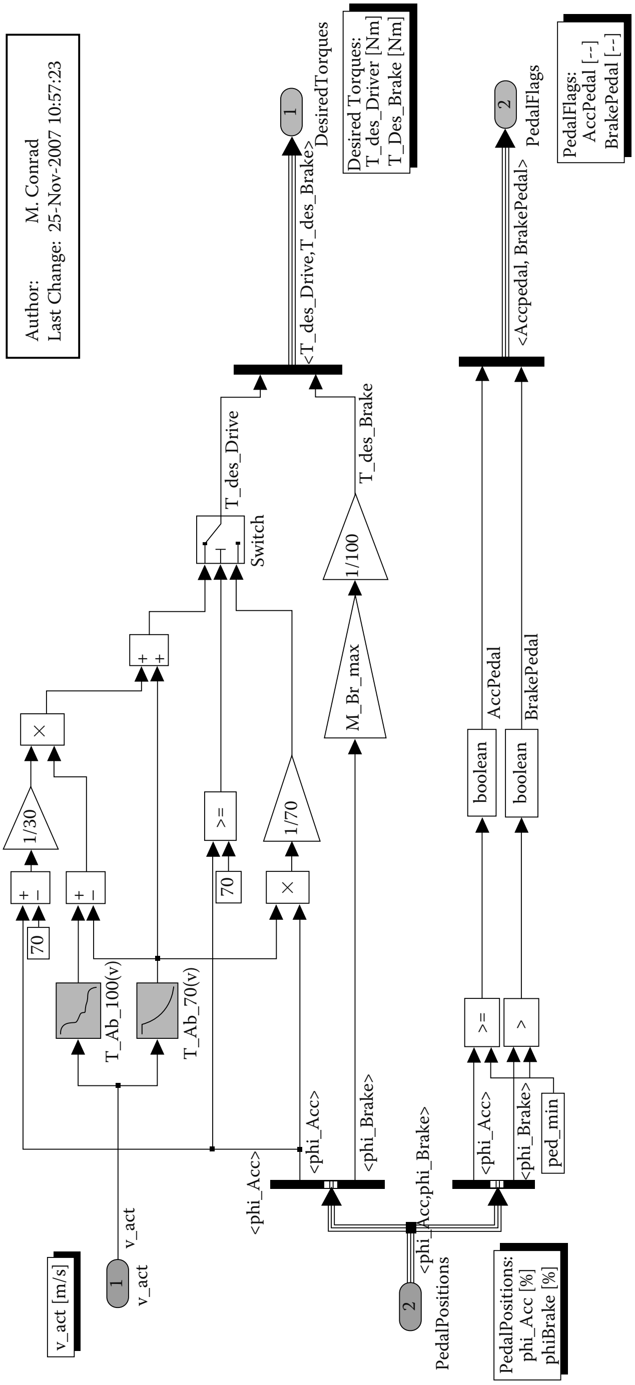


FIGURE 11.4 Model of the PedInt component.

TABLE 11.2 Interface Description of PedInt

Variable	↔	Unit	Range	Data Type
v_act	·	m/s	[−10, 70]	Real
phi_Brake	·	%	[0, 100]	Real
phi_Acc	·	%	[0, 100]	Real
T_des_Drive	·	Nm		Real
T_des_Brake	·	Nm		Real
AccPedal	·	—	{0, 1}	Bool
BrakePedal	·	—	{0, 1}	Bool

abstractions of individual inputs for testing purposes. The partitioning, which is graphically represented by means of a "classification tree," aims to achieve a definition of the individual "equivalence classes" in such a way that they behave homogeneously with respect to the detection of potential errors. That is, the test object behaves either correctly or erroneously for all the values of one class ("uniformity hypothesis").

A heuristic procedure has proved successful in approaching this ideal partitioning as much as possible in practice. The inputs' data types and ranges provide the first valuable clues to partitioning: where real-valued data types with established minimum and maximum values are concerned, it is possible, for example, to create a standard partitioning with a class each for the boundary values, for the value of zero and for those intervals in between. Similar "standard classifications" that are data-type-specific can also be utilized for other data types [CDS+02,Con04a,Con04b].

In general, the data-type-specific standard classifications are not detailed enough for a systematic test. They have to be refined or modified manually in order to approach partitioning according to the uniformity hypothesis. The quality of the specification and the tester's experience are crucial in this respect.

According to the standard partitioning, the pedal positions that can take values from the range of 0%–100% would be partitioned into three classes 0,]0, 100[, and 100. For the vehicle speed the five partitions −10,]−10,0[, 0,]0,70[, and 70 are obtained.

As v_act is only of minor importance for testing these subfunctions of PedInt that both flags calculate, the standard classification has not been modified there. On the other hand, the evaluation of the pedal positions recognizes a pedal as depressed only if it is activated above a certain threshold value ped_min. Therefore, the pedal values above and below the threshold should be considered separately because behavior is expected to differ. A class has also been added for the exact threshold value. In order to keep the classification-tree flexible, parameter names were partly used as class boundaries rather than fixed values. The result is a final partitioning of the pedal positions into the classes 0,]0, ped_min[, ped_min,]ped_min, 100[, and 100.

The partitioning into classes is visualized by means of a classification tree (Figure 11.5, middle part). The lower part of Figure 11.5 shows how the equivalence classes for the individual input variables partition the input domain of the entire test object.

The actual test scenarios are designed on the basis of the test input partitioning. The test scenarios describe the course of these inputs over time in a comprehensive, abstract manner. Therefore, the leaves of the classification tree are used to define the columns of a "combination table." In order to represent test scenarios in an abstract way, they are decomposed into individual test steps. These compose the rows of the combination table according to their temporal order (Figure 11.6, upper and middle parts).

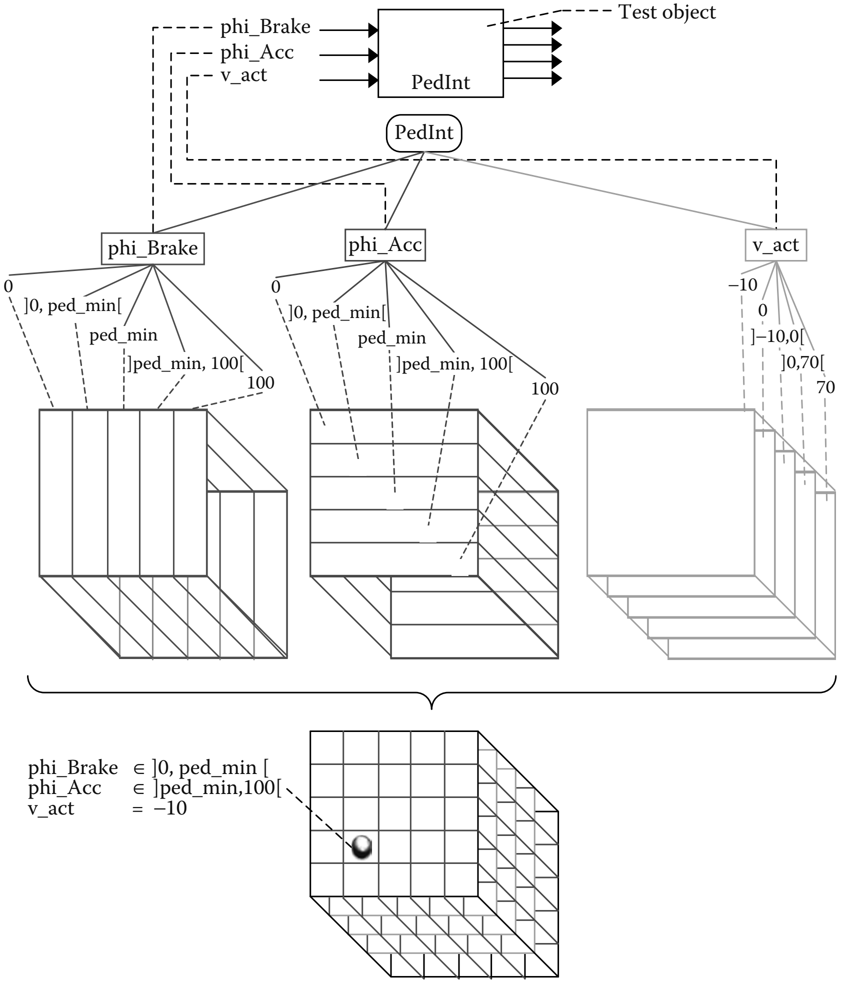


FIGURE 11.5 From test interfaces to classification trees.

The test inputs for each test step are defined by combining classes of different classifications from the classification tree. This is done by marking the appropriate tree elements in the main column of the combination table. This leads to a sequence of test input situations. The duration of each input situation can be defined by the annotation of a "time tag" in the rightmost column of the combination table (Figure 11.6, middle part).

Timed inputs, that is, changing the values of an input over time in successive test steps, can be represented by marking different classes of the classification corresponding to that input. The test input in the respective test step is thus restricted to the part-interval or single value of the marked class. The combination of the marked input classes of a test step determines the input of the test object at the respective time (Figure 11.6, middle part).

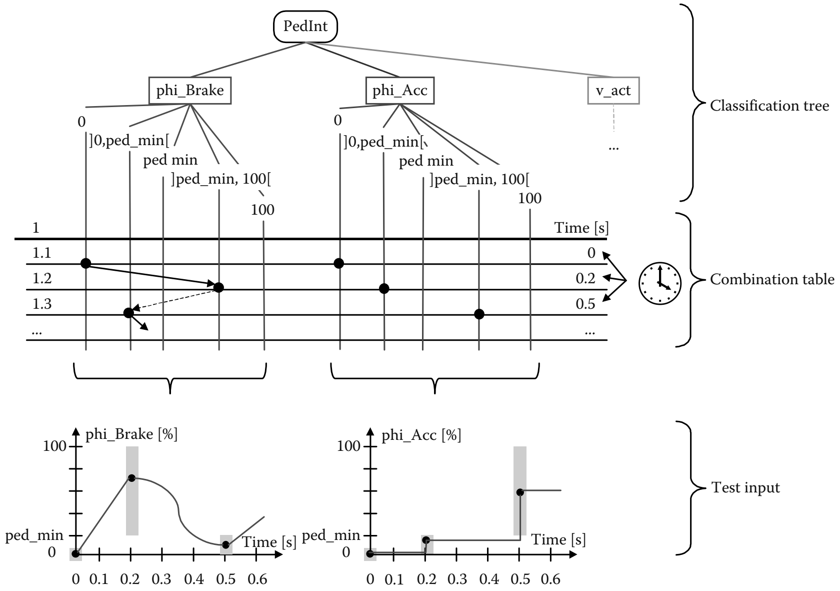


FIGURE 11.6 Defining test sequences.

TABLE 11.3 Assignment of Signal Shapes for Line Types

Signal Shape	Transition Type
Step	
Ramp	—————
Sine/spline	- - - - - - - - -

The intermediate values of the individual stimuli signals are defined by transitions between markings of consecutive test steps. Different transition types represent different signal shapes (e.g., ramp, step function, sine; Table 11.3). In this way, stimuli signals can be described in an abstract manner with the help of parameterized, stepwise defined functions (Figure 11.6, middle and lower parts).

Further test sequences can be described underneath the classification tree by repeating the procedure outlined above.

For the PedInt example in the first test scenario (Figure 11.7, test scenario #1), the aim is to investigate the recognition of pedal activation with specific values, each of which is kept constant for a certain period of time.

A second test scenario (Figure 11.7, test scenario #2) checks the hysteresis properties of the test object. In order to do this, the pedal positions are varied, in the form of a ramp, over the entire value area, each going upward and downward once. In doing so, classes are selected arbitrarily for v_act.

A third test scenario (Figure 11.7, test scenario #3) linearly ramps up the entire nonnegative speed interval in 2 s rhythm for different but constant pedal positions.

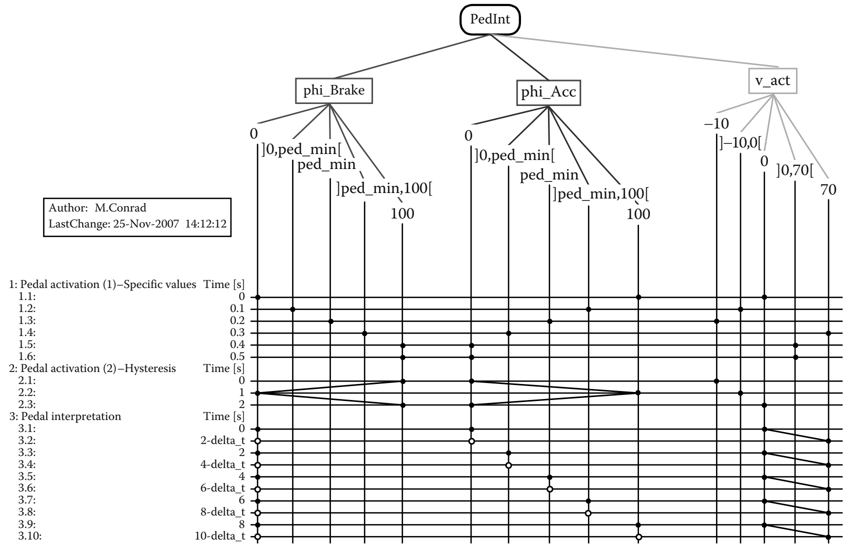


FIGURE 11.7 Classification tree with test scenarios for PedInt.

The definition of test scenarios on the basis of this input data portioning is a combinatorial task: Different combinations of input data classes are selected and sequenced over time.

After the design of test scenarios has been completed, it is necessary to check if they ensure sufficient test coverage. At the test design stage, CTM_{EMB} already allows the determination of different abstract coverage criteria on the basis of the classification tree and the test scenarios.

A "requirements coverage analysis" can verify whether all specified requirements that concern the test object are covered sufficiently by the test scenarios designed so far. In the course of the analysis, it is necessary to prove that every requirement is being checked by at least one test scenario and that the existing test scenarios are adequate to test the respective requirements.

Table 11.4 presents the coverage of requirements by means of those test scenarios that have been determined up to this point. The necessary requirements coverage has thus been fulfilled for the requirements SR-PI-01.1 to SR-PI-01.4. The higher-level requirement SR-PI-01 does not have to be tested separately, as it has already been tested implicitly by all of the derived requirements being tested.

TABLE 11.4 Traceability Matrix PedInt

	SR-PI-01	SR-PI-01.1	SR-PI-01.2	SR-PI-01.3	SR-PI-01.4
Test scenario #1	(✓)	✓		✓	
Test scenario #2	(✓)		✓		✓
Test scenario #3					

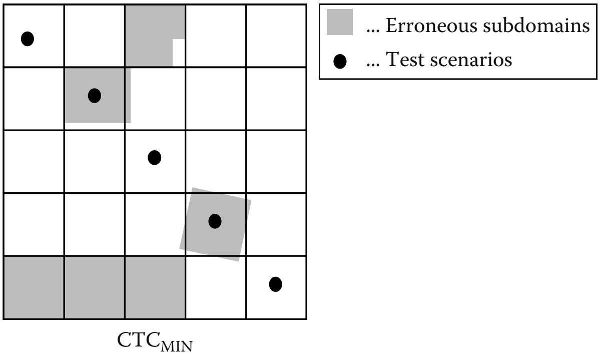


FIGURE 11.8 Minimum criterion, CTC$_{\mathrm{MIN}}$.

Furthermore, the CTM$_{\mathrm{EMB}}$ supports a "range coverage analysis." This analysis checks the sufficient consideration of all equivalence classes defined in the classification tree in the test scenarios. This check can be executed, according to the respective application case, by using different, so-called "classification-tree coverage criteria" (CTC) [GG93,LBE+04].

The "minimum criterion," CTC$_{\mathrm{MIN}}$ (Figure 11.8), requires every single class in the tree to be selected in at least one test step. The minimum criterion is normally accomplishable with a few test sequences. The error detection rate, however, is rather low.

The "maximum criterion," CTC$_{\mathrm{MAX}}$ (Figure 11.9), requires every possible class combination to be selected in at least one test step. The fulfillment of the maximum criterion should result in a high error detection rate. However, the problem with this criterion is a possible combinatorial explosion. This makes it impracticable when a large number of classes are involved.

The "n-wise combination criterion," CTC$_n$, presents a compromise. Here, it is necessary to ensure that every possible combination of n classes is selected in at least

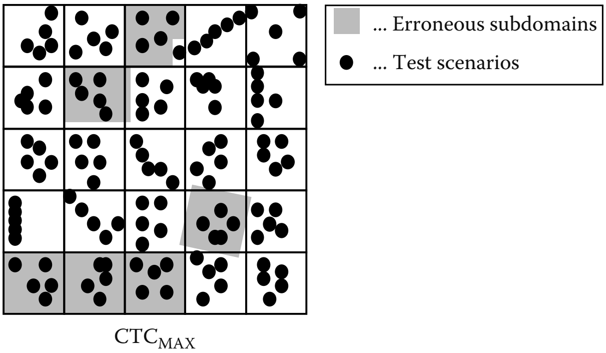


FIGURE 11.9 Maximum criterion, CTC$_{\mathrm{MAX}}$.

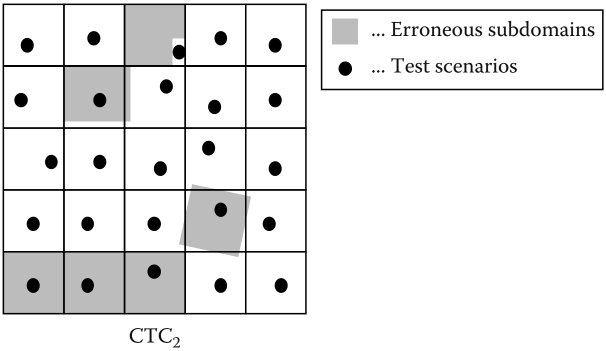


FIGURE 11.10 Pair-wise combination criterion, CTC_2.

one test step. For example, a pair-wise combination of classes, CTC_2 (Figure 11.10), is practicable.

The selection of appropriate criteria has to take place in a problem-specific way during test design. If the criteria defined beforehand have not been sufficiently fulfilled, additional test scenarios need to be added until the required criteria are reached.

In terms of data range coverage, the three determined test scenarios (Figure 11.7) fulfill the minimum criterion, CTC_{MIN}, but they do not fulfill any higher criteria, such as CTC_2 or even CTC_{MAX}. In practice, this means that further test scenarios would have to be added until the attainable classification-tree coverage criterion is also fulfilled.

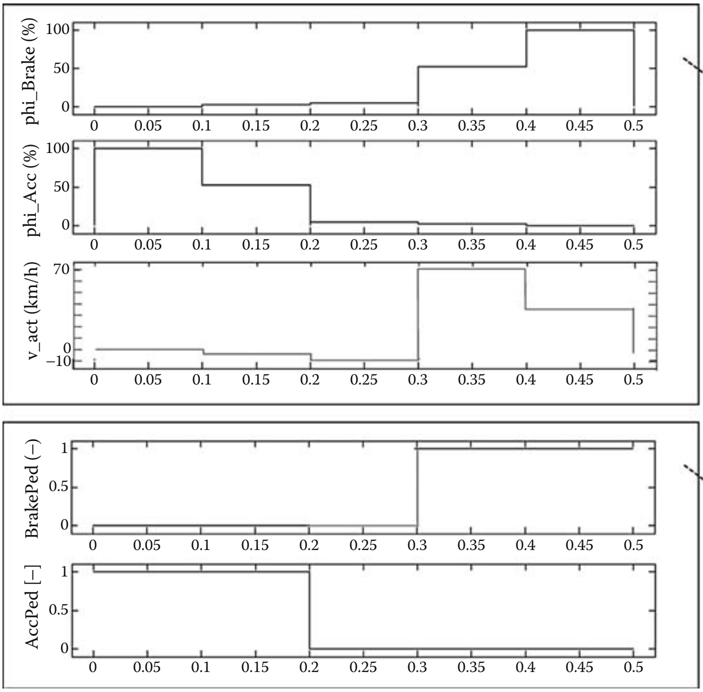
The test scenarios gained so far contain abstracted stimulus information because only equivalent classes, but no specific data have been used. Thus, in a further step, they need to be instantiated by the use of specific numbers.

The test scenarios defined in the combination table represent "signal corridors" for the individual input signals, within which the actual courses of input time series must be located. The boundary values of the equivalence classes constrain the input ranges at the respective test steps. This step is illustrated in the lower part of Figure 11.6.

On the basis of the uniformity hypothesis, any value within the marked interval can be selected when determining the values at the base points. Within the scope of this example the principle of mean value testing is being used, that is, in each case the mean values of the equivalence classes selected are used as test data (ped_min = 5%, Figure 11.11).

What has been presented so far is how to systematically determine and compactly describe test scenarios for the black-box test of automotive control software with the help of CTM_{EMB}. Requirements coverage and input data range coverage (classification-tree coverage) allow for the quality assessment of the determined test scenarios.

CTM_{EMB} can be deployed as a black-box test design technique, independently of an actual development process. The only prerequisites are the description of the test object's behavior and interface.



Test input

Expected output

FIGURE 11.11 Test data time series for PedInt (test scenario #1).

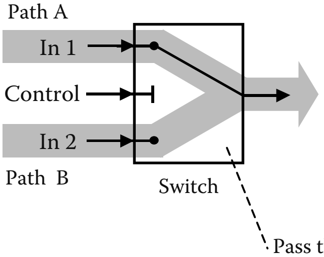
11.2.2.2 Structural Test Design: Decision Coverage Testing

In structural test design, the test cases are derived from the structure information on the test object. In the control flow-oriented test design, these result from the control flow of the test object. In the context of the model-based development, it is possible to both use the control flow of the model and that of the source code as a basis for this.

The test design's target in the control flow-oriented structure test is the coverage of certain items of the control flow graph. If the test object is a Simulink/Stateflow [SL,SF] model, it is possible to draw on the model coverage criterion "decision coverage" (DC) for the determination of the test objectives. DC examines items that represent decision points in a model, such as the switch blocks and Stateflow states. For each item, the different simulation paths through that item are to be covered [BCS+03].

Figure 11.12 shows, as an example, the different pathways through a switch block and lists the test objectives that have to be fulfilled in order to achieve full coverage.

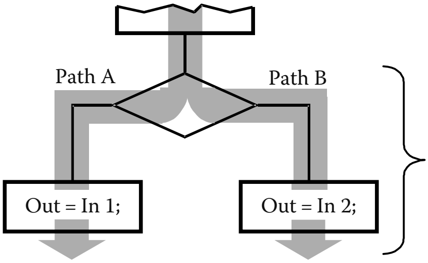
The only block in the PedInt model that is relevant for DC is the switch block in the upper, right part. It passes through the upper input, if phi_Acc is greater than or equal to 70, otherwise it passes through the lower input. In order to achieve full DC, phi_Acc must be less than 70 for some time and greater than or equal to 70 for some other time. Test scenario #1 satisfies this condition; there is no need for additional structural test scenarios in order to achieve DC on model level.



Pass through In 1 when
Control ≥ threshold;
otherwise, pass through In 2

Test objective	Test input specification
1 (Path A)	Control ≥ threshold
2 (Path B)	Control < threshold

FIGURE 11.12 DC on model level—pathways through a switch block and test objectives.



```
If (Control ≥ threshold){
   Out = In 1;
} else {
   Out = In 2;
}
```

Test objective	Test input specification
1 (Path A)	Control ≥ threshold
2 (Path B)	Control < threshold

FIGURE 11.13 Branch coverage on code level—control flow graph and test objectives.

A related criterion on code level is "branch coverage" (C_1). The objective of C_1-based test design is to execute each program branch of the source code at least once. Here, a program branch is roughly defined as a possible route from the program start, or from a branching of the control structure, to the next control structure or program end [BCS+03].

Figure 11.13 shows, as an example, the different branches emerging from an If-statement and lists the test objectives that have to be satisfied in order to achieve full C_1 coverage.

11.2.3 Exemplary Test Execution Techniques for Automotive Control Software

11.2.3.1 Back-to-Back Tests

Traditional, code-based software development usually produces only one "executable artifact;" the software itself. Due to the model evolution, model-based design, on the other hand, usually produces several executable artifacts (e.g., design model, implementation model, and generated code) for one and the same functionality. Moreover, most of these diverse artifacts can be tested in their entirety (aka system test) and/or partially (aka module test). In combining these "potential test objects" with different "execution platforms" (e.g., host PC, evaluation board) and "environments" (e.g., open-loop, closed-loop with simulated environment model, closed-loop with real environment), a multitude of so-called "test possibilities" arises [BN03,GS05]. For efficiency reasons, it is not advisable to make use of all existing test possibilities for a thorough model-based test. Rather a well-considered subset should be chosen.

Within the range of these test possibilities, a test suite is typically applied to different artifacts. This enables test approaches to be implemented, where the system reaction of one artifact can be used as a test oracle for the determination of the expected outputs of a different artifact. In particular, it is often possible to verify the correct transition from one model evolution stage to the next by "back-to-back tests between different artifacts" (equivalence testing).

If a test scenario is applied to different artifacts, it might be necessary to adapt it to the particular test object. As an example, the next section discusses how test scenarios are applied to model level tests.

11.2.3.2 Test Harness Generation for Model Tests

In order to stimulate a test object in the form of a Simulink/Stateflow model with the defined test scenarios, an infrastructure is required, which feeds the test inputs into the test object and captures the system reaction. This infrastructure can, for example, consist of a "model test harness," which itself again is a Simulink/Stateflow model.

When creating the test harness, a copy of the test object in a separate model is complemented by stimulation and signal logging blocks, so that an extended "executable model for the test execution," called test harness model emerges (Figure 11.14). The stimuli signals defined in the course of the test design are converted by the test harness into a representation suitable for the model test and stimulate the inputs of the test object. At the same time, on the side of the output, the logging of the relevant output signals of the test object and a reconversion into a Simulink-independent representation are carried out. If a closed-loop test execution is desired, it is necessary to integrate corresponding model components (parts of plant and environment model) to model the feedback loop and to add the related signal flows.

In the case of "indirect stimulation" model parts for the transformation of the signals contained in the stimulation blocks into the input parameters of the test object have to be inserted [Con04a,Con04b].

Thus, it is possible, for example, to add up the test data for the actual value of a signal and the difference between the set-point value and the actual value of the signal by using a sum block in order to calculate the set-point value of this signal, if this is needed as the input for the test object. Other applications of an indirect stimulation include range conversions (e.g., m/s to mph) or the transformation of pedal travel values into corresponding torque requests via look-up tables.

Analogously to the indirect stimulation, it may make sense "to capture derived signals" instead of outputs of test object or in addition to them. Examples for this are again range conversions or the logging of filtered signals.

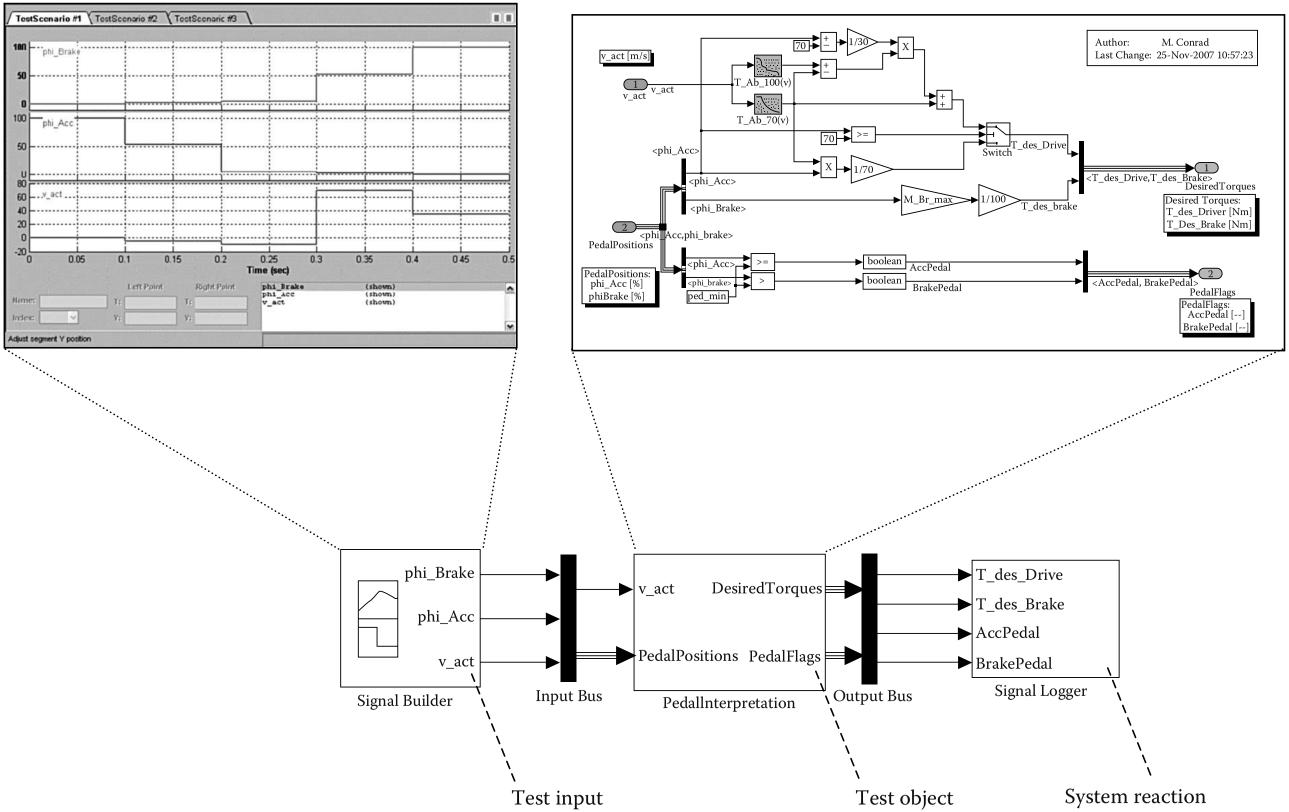


FIGURE 11.14 Model test harness for PedInt.

Another use case for capturing derived signals is the logging of "watchdog" outputs. Watchdogs are model parts used to monitor model signals and check properties between them. The Boolean output of a watchdog that asserts when the signals to be monitored violate a specified property, is a derived signal that can be logged for further analysis.

Test infrastructures for other representation forms of the test object, that is, for software testing or embedded system testing, are also created via the principles described above. However, specifics of the respective execution platforms and environments have to be taken into account.

11.2.4 Exemplary Test Evaluation Techniques for Automotive Control Software

11.2.4.1 Signal Comparison for Regression and Back-to-Back Tests

In the case of timed test scenarios, that is, test execution results in timed test outputs, the system reaction is available in the form of timed signals. Consequently, timed system reactions (output time series) have to be compared with timed reference behavior (reference time series, golden references, or baselines) in the course of test evaluation. Accordingly the essence of the test evaluation in this case is to carry out "signal comparisons" between output time series $o^{*\prime}(t)$ and reference time series $o^{*}(t)$. Since exact equality in practice can only be expected in special cases, it is necessary to use suitable "robust signal comparison techniques," which are able to tolerate deviations in the time and value domain between the time series to be compared.

"Manual signal comparison" can be carried out by means of a visual assessment of the output time series and reference time series plotted into one and the same diagram. Such a check will not necessarily focus on absolute equality but rather on a certain similarity: signals are considered to be similar if the signal plots are sufficiently close together. This kind of visual check is, however, not only highly subjective but also, depending on the number and size of the signals to be compared, highly prone to error and, at the least, very time consuming. It also requires experienced testers.

"Automated signal comparison" aims to automatically assess whether or not the test object produces a system reaction sufficiently similar to a former approved baseline. Accordingly, the test evaluation judges whether $o^{*\prime}(t)$ and $o^{*}(t)$ are similar. If no similarity between the time series is automatically recognizable, deviations that have occurred need to be detected and localized as far as possible. In this case, suitable information to facilitate further manual assessment should be made available to the human tester.

Popular methods for automated signal comparison include absolute, relative, and slope-dependent differences.

1. "Absolute difference:"

$$\text{absDiff}_j(t_i) = |o_j^{*}(t_i) - o_j^{*\prime}(t_i)|$$

2. "Relative difference:"

$$\text{relDiff}_j(t_i) = \frac{|o_j^*(t_i) - o_j^{*\prime}(t_i)|}{\sqrt{|o_j^*(t_i)|}\sqrt{|o_j^{*\prime}(t_i)|}}$$

3. "Slope-dependent difference:"

$$\text{slopeDiff}_j(t_i) = \frac{|o_j^*(t_i) - o_j^{*\prime}(t_i)|}{\sqrt{\left|\frac{1}{2\Delta t}(o_j^*(t_{i-1}) - o_j^*(t_{i+1}))\right|}\sqrt{\left|\frac{1}{2\Delta t}(o_j^{*\prime}(t_{i-1}) - o_j^{*\prime}(t_{i+1}))\right|}}$$

Given a test object with an output variable o_j under consideration, in the above formulas $o_j^{*\prime}(t)$ denotes the time series of expected/reference outputs (defined during test design) and $o_j^*(t)$ the time series of actual outputs (captured during test execution), respectively.

These "difference criteria" are suitable for the detection of simple deviations in the value domain, but they do not tolerate (partial) deviations in the time domain between the signals to be compared.

In the case of large amplitudes or slopes, relative and slope-dependent differences overweight segments with small amplitudes or slopes, in order to tolerate deviations. In segments with zero crossings even very small amplitude deviations will cause unacceptably high differences. To address this issue, the standard methods for relative and slope-dependent differences can be modified as proposed in Ref. [CSW06]:

4. "Modified relative difference:"

$$\begin{aligned}\text{modRelDiff}_j(t_i) &= \min(\text{relDiff}_j(t_i), \text{absDiff}_j(t_i))\\ &= \frac{|o_j^*(t_i) - o_j^{*\prime}(t_i)|}{\max\left(\sqrt{|o_j^*(t_i)||o_j^{*\prime}(t_i)|}, 1\right)}\end{aligned}$$

5. "Modified slope-dependent difference:"

$$\begin{aligned}&\text{mod SlopeDiff}_j(t_i) = \min(\text{slopeDiff}_j(t_i), \text{absDiff}_j(t_i))\\ &= \frac{|o_j^*(t_i) - o_j^{*\prime}(t_i)|}{\max\left(\sqrt{\left|\frac{1}{2\Delta t}(o_j^*(t_{i-1}) - o_j^*(t_{i+1}))\right|}\sqrt{\left|\frac{1}{2\Delta t}(o_j^{*\prime}(t_{i-1}) - o_j^{*\prime}(t_{i+1}))\right|}, 1\right)}\end{aligned}$$

The modifications represented by these formulas lend weight to deviations at large amplitudes or slopes without overestimating the parts of small amplitude values or slopes.

The selected difference criteria is calculated for all time steps t_i of the given time series. To produce the test verdict, the maximum difference obtained this way is then compared with an "acceptance threshold" (max. tolerated deviation).

In order to evaluate test scenario #1 for the PedInt component (Section 11.2.2.1), the system reaction regarding the two pedal flags has to be compared with the

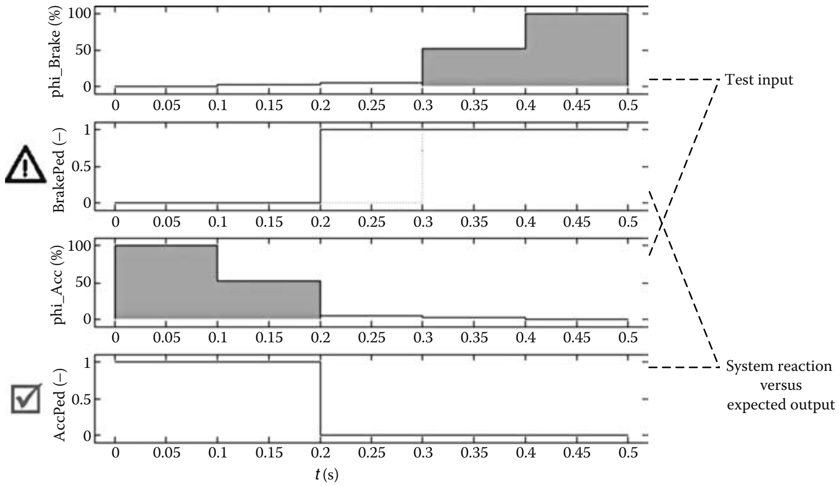


FIGURE 11.15 Test evaluation PedInt (test scenario #1).

expected outputs. Since the AccPedal and BrakePedal are Boolean values, absDiff with a tolerated deviation of 0 was chosen.

The top plot in Figure 11.15 shows the brake-pedal travel over time. Segments in which it is greater than the threshold value ped_min are colorized. The subjacent plot shows the expected (solid line) and the actual (dotted line) values of the brake-pedal flag. As they are not identical in the area between 0.2 and 0.3, the brake-pedal recognition is not in accordance with the specification. In analogy, the two lower plots are concerned with the accelerator pedal activation recognition. Here, it is functioning correctly. Actual and expected signal responses are in agreement.

A reason for this erroneous behavior could be that the implemented comparison algorithm (Figure 11.4), which compares the brake pedal travel phi_Brake with the respective threshold value ped_min, incorrectly checks for > = instead of >.

More powerful procedures for robust signal comparison are a current research topic [DSS+01,HT01,RWS+01,SWM02,WGP02,CS03].

A certain prevalence in the application field has a combination of the "difference matrix procedure" and downstream difference calculation, which is described in Refs. [WCF+02,CFP03,CSW06].

This two-staged signal comparison technique first preprocesses the two time series by using the difference matrix algorithm in order to access temporal deviations. Second, the preprocessed timed signals are compared according to the above-mentioned elementary difference criteria to evaluate the differences in the value domain. This way, temporal shifts and deviations in amplitude are identified and judged separately.

When using the difference matrix procedure for preprocessing, the system reaction is adjusted to the expected output by stretching or compressing it temporally. The extent to which the signal has to be stretched or compressed indicates the temporal displacement. The core idea is as follows: Suppose we have to investigate the similarity between two signals that had been shifted in time with respect to each other. We would shift one of the signals toward the other and would examine the remaining deviation. Mathematically speaking, moving one of the signals toward the other corresponds to a special "reparameterization" (temporal reordering) of the signal. The difference matrix procedure looks for a general reparameterization, such that one signal matches the other as much as possible. Thus, possible local shifts and compressions are taken into account.

Therefore the algorithm calculates a suitable reparameterization (temporal reordering) $\gamma : [1, t_{\max}] \rightarrow [1, t_{\max}]$ of the system reaction $o_j^*(t)$ such that it approximates the reference signal $o_j^{*\prime}(t)$ as well as possible, that is, best match, so that $o_j^*(\gamma(t)) \approx o_j^{*\prime}(t)$ is valid. When doing this, the temporal order of the sample points in the time series must be respected.

If the time deviations discovered by the difference matrix procedure do not exceed a given threshold, an assessment is then made as to the similarity among the reparameterized signals using the difference criteria discussed above.

There are also numerous test evaluation procedures for test objects that fulfill specific prerequisites. Watkins [Wat82], for example, describes a procedure for control software, which calculates sufficiently continuous functions, in which future system outputs can be estimated and monitored on the basis of previous outputs. In practice, it is necessary to assemble evaluation procedures that are suitable for the specific problem.

11.3 Testing in the Development Process

As outlined in the introduction it is essential to adapt the overall testing process according to the software development paradigm used. Therefore, in the following testing processes for model- and code-based development are outlined. Substantial differences between these development paradigms, with respect to testing, arise from the different potential test objects taken into consideration as well as from the typical test levels. Figure 11.16 gives an overview on test objects and test levels for model- and

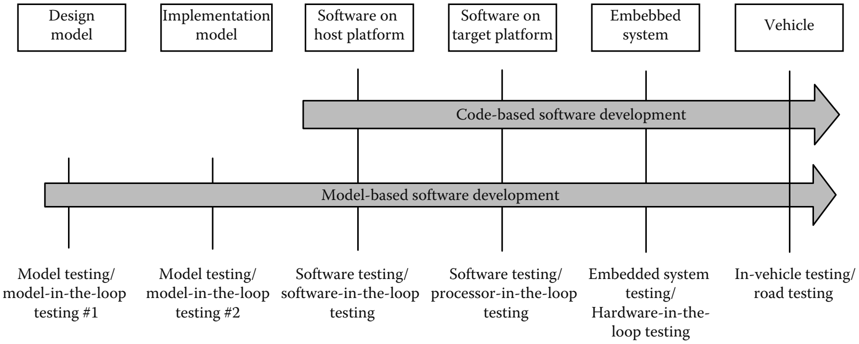

Design model
Implementation model
Software on host platform
Software on target platform
Embebbed system
Vehicle
Code-based software development
Model-based software development
Model testing/ model-in-the-loop testing #1
Model testing/ model-in-the-loop testing #2
Software testing/ software-in-the-loop testing
Software testing/ processor-in-the-loop testing
Embedded system testing/ Hardware-in-the-loop testing
In-vehicle testing/ road testing

FIGURE 11.16 Test objects and test levels for model- and code-based control software development (overview).

code-based software development, respectively. Sections 11.3.1 and 11.3.2 will detail this overview.

Specific challenges for testing in the automotive context arise out of the distributed development with the car manufacturer (original equipment manufacturer [OEM]) on the one hand and the supplier on the other hand, regardless of whether the development follows the code-based or the model-based paradigm. Section 11.3.3 discusses the relationship between the OEM and its suppliers within the testing process.

11.3.1 Testing in a Code-Based Development Process

11.3.1.1 Test Levels for Code-Based Testing

Testing in a code-based development process is fairly consistent with the typical test phases described in several textbooks on test theory [Mye79,Lig92].

The individual software components (modules) are the first executable and thus dynamically testable artifacts that are produced. During "software component testing," each module is tested individually and the component interfaces are checked for consistency with the design specifications. Although software component testing is generally considered as the responsibility of the software developer, the results are part of the whole project and thus test management should be aware of them.

After the individual modules have been tested, the software system is assembled and tested according to the integration strategy defined (Section 11.4.4). So during "software integration testing," a number of software components representing subsystems are to be tested before the entire software is completely integrated and ready for testing as a whole. The number of integration levels and the sequence of incorporation of the individual modules or subsystems are defined in the integration strategy. Software integration testing aims, for example, at revealing errors resulting from incorrect component interface implementation, incorrect error handling, improper control, and sequencing of components.

In theory, both software component testing and software integration testing can be executed on a host as well as on a target platform. In practice, however, both are typically executed on the host computer since the intercomponent interfaces can be stimulated more easily here (Figure 11.17).

"Software system testing" aims at testing the integrated software as a whole. Software system testing can be performed on the host (software-in-the-loop testing [SIL]) as well as on a target processor (processor-in-the-loop testing [PIL]). If a target processor is available within this phase it is recommended to run specific tests with respect to timing aspects on this processor.

After the software has been tested thoroughly, the next integration step has to be applied, that means the integration of the software into the embedded system, that is, the target ECU. This integration step is followed by a "system component testing" or "ECU testing."

According to the progress of the hardware development in the actual project, the system test is executed by using the currently available sample of the control unit (A, B, C samples).

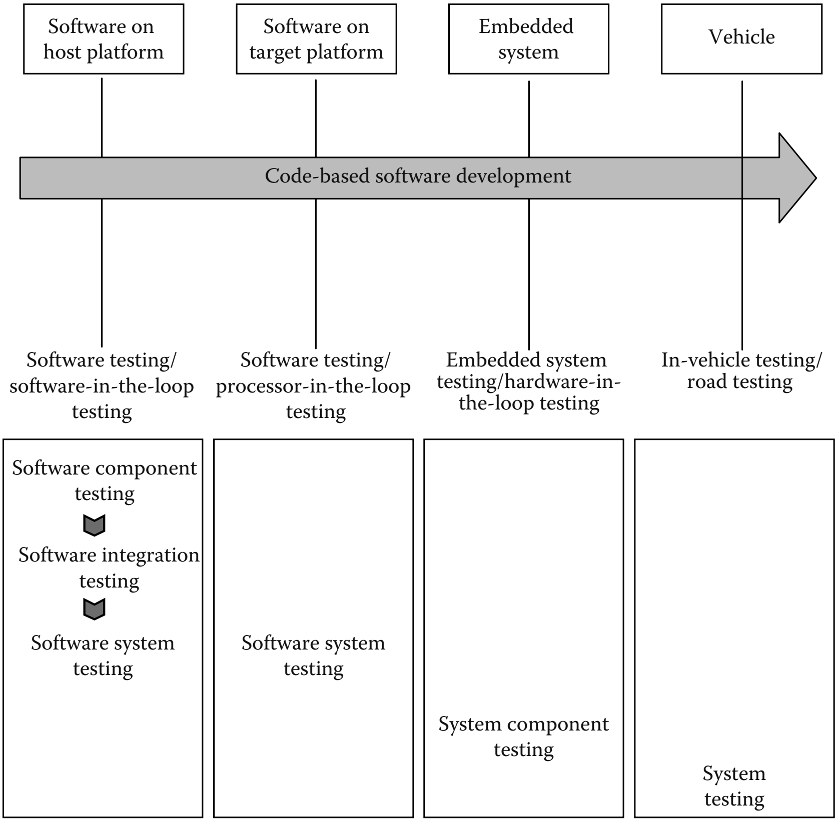

Software on host platform
Software on target platform
Embedded system
Vehicle
Code-based software development
Software testing/ software-in-the-loop testing
Software testing/ processor-in-the-loop testing
Embedded system testing/hardware-in-the-loop testing
In-vehicle testing/ road testing
Software component testing
Software integration testing
Software system testing
Software system testing
System component testing
System testing

FIGURE 11.17 Test levels for code-based development (detailed view).

Finally, the embedded system is integrated into the car and tested by means of "in-vehicle-tests".

11.3.1.2 Test Strategy for Code-Based Testing

Applying a single testing technique usually fails in reaching satisfactory test coverage. In practice, a number of complementary test techniques are carefully combined to form a "test strategy." Powerful test strategies comprise combinations of functional and structural test design techniques. The term "effective test strategy" was introduced by Grimm [Gri95] to describe this approach.

11.3.2 Testing in a Model-Based Development Process

11.3.2.1 Test Levels for Model-Based Testing

Testing in a model-based development process extends the scope known from the original testing theories. Utilization of the model as a test object permits a distinctly earlier start of test execution compared to code-based development. The functional model to be tested usually appears in different evolution stages. The two variations that can be found in almost every model-based project are the "behavioral" or "design model" and the "implementation model." Whereas the design model focuses on depicting the intended behavior, the implementation model contains all realization aspects that are the prerequisite for the coding activities.

Because of the early presence of executable controller models, it is already possible for the function developer to begin proving and testing with different techniques in the modeling phase. Errors can thus be detected early and removed at low cost. The spectrum of usable techniques ranges from interactive ad hoc simulations to systematic testing of the model.

The test levels for code-based development (Section 11.3.1.1) also apply for model-based development. Software component testing, and software integration testing can be carried out a bit more flexibly in respect of the execution platform, since automatic code generators typically support both, a host as well as a target platform. For practical reasons, integration typically takes place on model level, so that on software level commonly only individual components and/or the entire application software is tested, but not the different integration steps (Figure 11.18). The main advantage of model-based testing here is the existence of additional test levels on model level that can be used to test the application and to increase its maturity early in the development, before any code has been produced.

In analogy to the software test levels built upon each other, "model component testing," "model integration testing," and "model system testing" can be addressed.

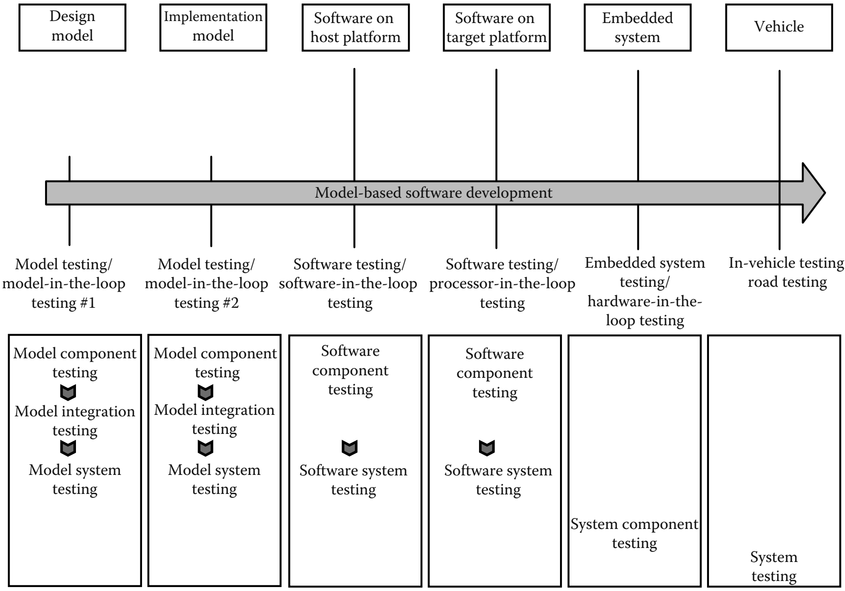

Design model
Implementation model
Software on host platform
Software on target platform
Embedded system
Vehicle
Model-based software development
Model testing/ model-in-the-loop testing #1
Model testing/ model-in-the-loop testing #2
Software testing/ software-in-the-loop testing
Software testing/ processor-in-the-loop testing
Embedded system testing/ hardware-in-the-loop testing
In-vehicle testing/ road testing
Model component testing
Model integration testing
Model system testing
Model component testing
Model integration testing
Model system testing
Software component testing
Software system testing
Software component testing
Software system testing
System component testing
System testing

FIGURE 11.18 Test levels for model-based development (detailed view).

For model level tests, both the design as well as the implementation model can be utilized.

If a plant model is included in the test, for example, testing is performed in a closed-loop manner, this is referred to as “model-in-the-loop test” (MIL). A similar terminology applies to the software level tests. Closed-loop tests on a host platform are termed “software-in-the loop tests” (SIL), closed-loop tests on a target platform are named “processor-in-the loop tests” (PIL).

The test procedure described here serves to test the controller to be developed and therefore focuses on the controller model. Irrespective of this, the quality of the vehicle and environmental models also obviously need to be safeguarded.

11.3.2.2 Test Strategy for Model-Based Testing

Test techniques for model-based testing [Con01,BN03] are, as a rule, adaptations of conventional software testing techniques or domain-specific analysis/testing techniques.

But applying test design techniques that employ the model as a test basis may result in different test scenarios, due to the special kind of representation and structuring as well as the different abstraction level. Furthermore, the model-based development approach allows a number of test techniques to be employed at an earlier stage.

Again, as the use of a single test technique is usually not sufficient for a thorough test, techniques that complement each other must be combined suitably. It is the aim of an effective test strategy to provide an appropriate combination of different testing procedures guaranteeing a high error detection probability. An effective test strategy for model-based testing has to take the specifics of model-based development into account and appropriately consider the existence of an executable model and thus exceed a general test strategy in two points. It should consider:

- Test design techniques in which the executable model of the software acts as a test basis
- Different representations (evolution stages) of the test object that typically appear in the context of model-based development

A combination of functional and structural test design techniques on model level with the subsequent reuse of the test scenarios thus determined has proven successful (Figure 11.19).

The emphasis of such a test strategy is on the systematic design of test scenarios based on the functional specification, the interfaces, and the executable model of the embedded software. In addition, an appropriate structural test criterion is defined on model level with which the quality of the tests that was determined in this way can be assessed. If necessary, additional test scenarios need to be defined until the selected structural test criterion has been fulfilled. Once sufficient test coverage on the model level has thus been ensured, the functional and structural test scenarios can be reused in the context of back-to-back tests for testing the software and the embedded system to prove the functional equivalence between the executable model and the representational forms deducted from this.

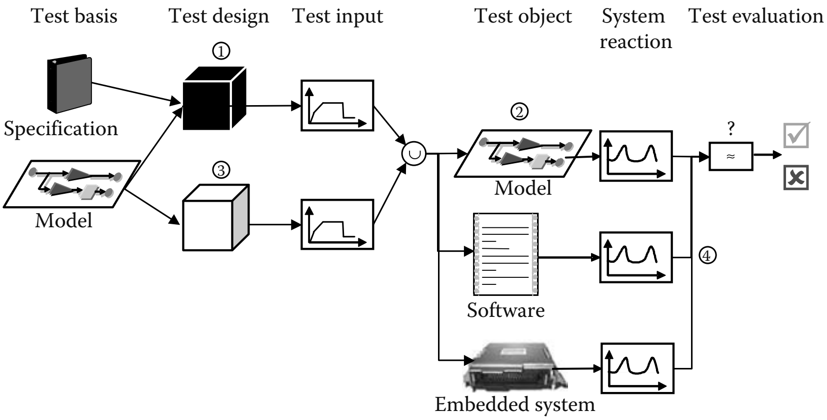


FIGURE 11.19 Effective test strategy for model-based testing.

In detail, the procedure is as follows:

1. *Systematic functional testing on model level*: Initially, functionally oriented test scenarios are systematically derived from the functional specification, the interfaces, and the executable model at an early stage of the development process (Sections 11.2.1.1 and 11.2.2.1).

   The determination of test scenarios is to be continued until sufficient requirements and value range coverage have been achieved. The detected test scenarios are then to be executed in the context of a model test. The model reaction to the test scenarios has to be recorded.

2. *Monitoring of model coverage*: It is useful to measure structural coverage on model level since this coverage can be determined before the actual software exists. This way, early statements about the structural coverage of the test object can be provided.

   The model coverage that was achieved with the test scenarios that were identified in step 1 (and in step 3, if applicable) has to be measured. We can proceed to step 4 once sufficient structural model coverage has been achieved. Otherwise we need to proceed with step 3.

3. *Structural testing on model level*: Should a sufficient model coverage not yet have been achieved, the model elements that have not been covered have to be identified and test scenarios have to be specifically created for the coverage of these model parts and have to be added to the existing test suite (Sections 11.2.1.1 and 11.2.2.2). Step 2 will subsequently have to be executed again. This procedure needs to be continued until sufficient model coverage has been achieved.

   The identified test scenarios are used to test the design and/or implementation model within the model test. The system reaction of the model resulting from stimulation with these test scenarios will have to be recorded.

4. *Execution of back-to-back tests*: with the software and the embedded system: If the software or the embedded system is available in the further development process, the test scenarios (that emerged in steps 1 and 3) are to be repeated on the software and/or the embedded system. The system reaction again is to be recorded and compared with the model reaction (Section 11.2.3.1). In case of sufficient similarity of the system reactions (functional equivalence), we can assume that the transformation of the model in C code and its embedding into the control unit has occurred error-free.

The effective test strategy resulting from steps 1 to 4 for model-based testing is depicted schematically in Figure 11.19. Selection of the test design techniques in steps 1 and 3, the structural test criterion in step 2, as well as the comparative procedures in step 4 is to be adapted project-specifically if necessary.

The systematic design of test scenarios based on functional as well as structural aspects makes an effective exposure of a variety of software-related errors possible. A sufficient test coverage is guaranteed in the context of the effective test strategy by a combination of different (e.g., requirement-oriented, data area-oriented, and [model] structure-oriented) coverage criteria.

The combination of functional and structural test design techniques on model level defined by the effective test strategy for model-based testing with a subsequent reuse of the detected test scenarios for testing of the software and the embedded system clearly brings out one of the main advantages of the model-based approach: being able to already systematically test the precursory stages of the embedded application software. Test activities can thus be shifted to earlier development phases and therefore decrease the error correction costs for those errors that are found early by model testing.

11.3.3 Interface and Interaction between OEM and Supplier

A characteristic feature of automotive controls development [SZ05] is the need to consider the relation between the OEM and its various suppliers. It influences the testing process such that the way of work share on the testing side obviously depends on the general model of cooperation.

A variety of cooperation models can be found in practice (Figure 11.20): On one side there is the traditional approach (type A), that is, the manufacturer specifies the system or component and the entire development is carried out by the supplier. On the other side, the contrary approach (type C), which becomes increasingly popular, is characterized by the fact that the complete application software development is carried out by the manufacturer and the supplier only provides the ECU hardware and some basic software. Many intermediate variants (type B), which are characterized by cooperative software development also exist. This kind of a more flexible interface between OEM and supplier enables the manufacturer to design new competitive functionality and keep the responsibility as well as the intellectual properties over the whole development cycle.

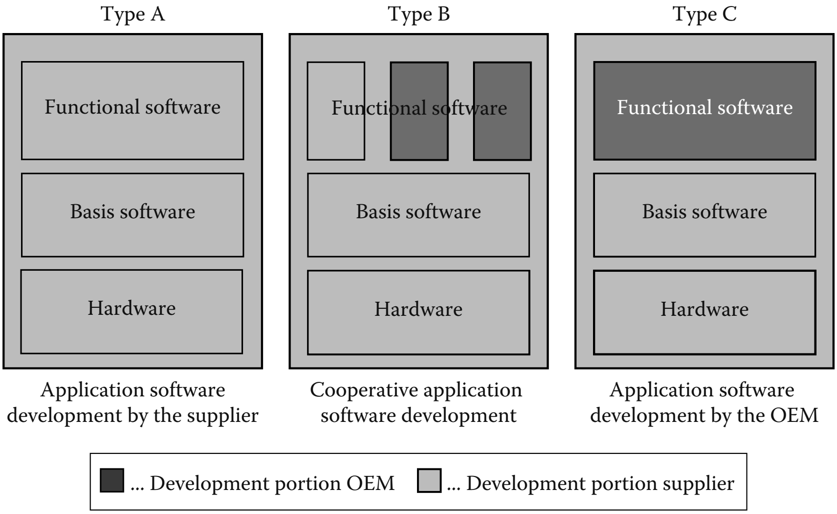

Type A
Type B
Type C
Functional software
Basis software
Hardware
Functional software
Basis software
Hardware
Functional software
Basis software
Hardware
Application software development by the supplier
Cooperative application software development
Application software development by the OEM
■ ... Development portion OEM
□ ... Development portion supplier

FIGURE 11.20 Types of cooperation within functional software development.

As far as testing is concerned, the OEM–supplier relationship causes that some test levels (Figure 11.18) are carried out on the OEM side and others on the supplier's side. In case of type A development, the majority of the test levels is performed by the supplier and the first time the OEM is involved is embedded system testing/HIL testing (Figure 11.21).

In case of types B or C development on the contrary, both development partners contribute to the testing process from the beginning up to the embedded system testing/HIL testing level. Figure 11.22 shows this relation for the example of a model-based software development and under the premise that all possible test levels are applied.

Furthermore, the interaction between development partners requires an additional test level on the manufacturer's side, the so-called "acceptance testing." Regardless whether the interface between the cooperation partners is defined one way or the other, the manufacturer has to apply an acceptance test suite on the development artifact that is provided by the supplier.

11.4 Test Planning

To ensure systematic and efficient testing, the different test levels and activities need to be coordinated and planned carefully. The aim of "test planning" is to determine the scope, procedure, resources, and schedule of the intended test levels and activities. The results of the test planning process are recorded in a *test plan* document.

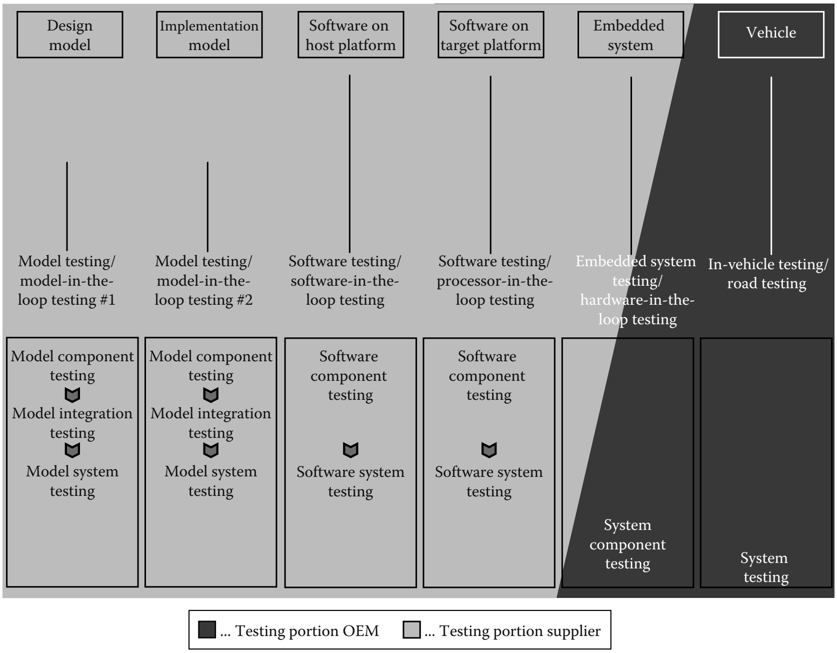

Design model
Implementation model
Software on host platform
Software on target platform
Embedded system
Vehicle
Model testing/ model-in-the-loop testing #1
Model testing/ model-in-the-loop testing #2
Software testing/ software-in-the-loop testing
Software testing/ processor-in-the-loop testing
Embedded system testing/ hardware-in-the-loop testing
In-vehicle testing/ road testing
Model component testing
Model integration testing
Model system testing
Model component testing
Model integration testing
Model system testing
Software component testing
Software system testing
Software component testing
Software system testing
System component testing
System testing
■ ... Testing portion OEM
□ ... Testing portion supplier

FIGURE 11.21 Interfaces in testing responsibilities for type A development.

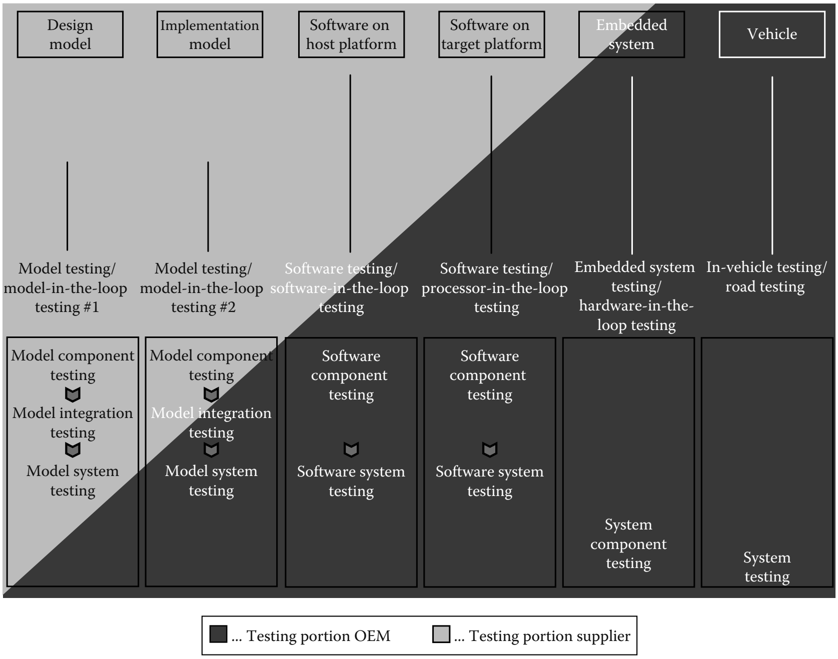

Design model
Implementation model
Software on host platform
Software on target platform
Embedded system
Vehicle
Model testing/ model-in-the-loop testing #1
Model testing/ model-in-the-loop testing #2
Software testing/ software-in-the-loop testing
Software testing/ processor-in-the-loop testing
Embedded system testing/ hardware-in-the-loop testing
In-vehicle testing/ road testing
Model component testing
Model integration testing
Model system testing
Model component testing
Model integration testing
Model system testing
Software component testing
Software system testing
Software component testing
Software system testing
System component testing
System testing
■ ... Testing portion OEM
□ ... Testing portion supplier

FIGURE 11.22 Interfaces in testing responsibilities for type B development.

11.4.1 Creating a Test Plan

The test plan determines at least

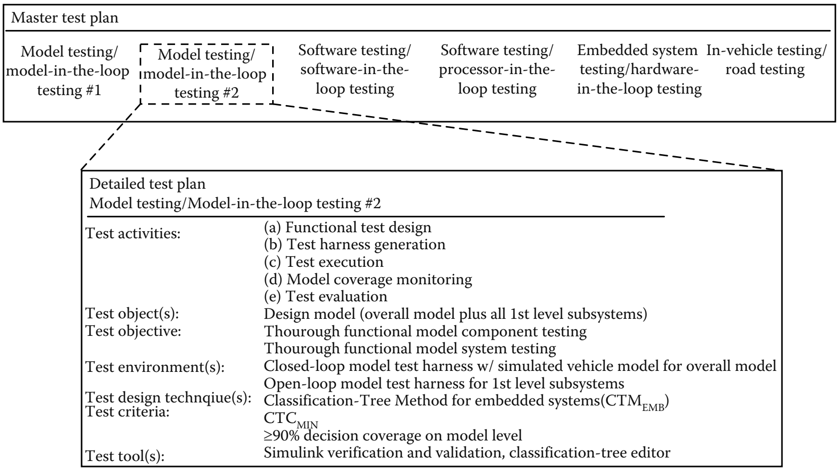
- Selection and combination of individual testing techniques (e.g., CTM $_{\mathrm{EMB}}$ and DC testing)
- Order in which they are applied to subcomponents within the overall system (e.g., top down or bottom up)
- Interaction between the different test levels (e.g., MIL, SIL)

In order to avoid both, test gaps and redundant tests in the multitude of test levels, and to enable a reuse of test scenarios that is as extensive as feasible, the tests in the individual levels need to be harmonized with respect to objectives, testing techniques, and tools to be used. In particular, it needs to be clearly documented what types of errors are addressed in the individual test levels and up to what depth the tests are to be carried out (test criteria).

The results of these considerations are to be documented in a "two-stage test plan" (Figure 11.23). Aspects spanning multiple test levels are pooled together in a "master test plan" that forms the basis for detailed test plans for the individual test levels. The master test plan establishes the coherence between individual test levels [BN03].

Detailed planning takes place separately for each test level and is recorded in "detailed test plans" for the individual levels. A test plan should contain information such as testing activities, test objects, test design technique(s), test environments, test criteria, required resources, etc. (Figure 11.23). Creating test plans can be facilitated by the use of test plan templates.

The test object(s) section of a detailed test plan identifies all test objects to be worked on in that test level as well as the corresponding test objectives. Based on the detailed test plan the tester conducts the different test activities for each of the listed test objects.


Master test plan
Model testing/ model-in-the-loop testing #1
Model testing/ model-in-the-loop testing #2
Software testing/ software-in-the-loop testing
Software testing/ processor-in-the-loop testing
Embedded system testing/hardware-in-the-loop testing
In-vehicle testing/ road testing

Detailed test plan
Model testing/Model-in-the-loop testing #2

Test activities:	(a) Functional test design (b) Test harness generation (c) Test execution (d) Model coverage monitoring (e) Test evaluation
Test object(s):	Design model (overall model plus all 1st level subsystems)
Test objective:	Thourough functional model component testing Thourough functional model system testing
Test environment(s):	Closed-loop model test harness w/ simulated vehicle model for overall model Open-loop model test harness for 1st level subsystems
Test design technqiue(s):	Classification-Tree Method for embedded systems(CTM$_{\mathrm{EMB}}$)
Test criteria:	CTC$_{\mathrm{MIN}}$ ≥90% decision coverage on model level
Test tool(s):	Simulink verification and validation, classification-tree editor

FIGURE 11.23 Test planning by two-stage test plans.

In order to trace the progress of the test, the results of the test activities should be recorded within an overall "test documentation." Thereby, the test documentation should reflect the structure of the test plan.

Some common activities to be carried out during test planning and topics to be considered are discussed in Sections 11.4.2 through 11.4.5.

11.4.2 Selection of Test Levels

The potential test levels depend on the project's development paradigm and life cycle. Typical test levels for model- and code-based development have been discussed in Sections 11.3.2.1 and 11.3.1.1, respectively.

Part of master test planning is, among other things, the determination of test levels applicable to the specific development context. The availability of evaluation boards is, for example, a precondition for PIL testing. On the other hand, it is not efficient to utilize all potential test levels. As an example, if design and implementation models are to be tested, one could consider carrying out model system testing for both models but cutting out unit tests for the design model. The integration strategy (Section 11.4.4) also influences the relevant test levels. In case of a big bang integration of models, for example, model integration testing is no longer useful.

The test strategy to be achieved (Sections 11.3.1.2 and 11.3.2.2) by the selected test levels should be documented within the master test plan. For each test level that is applicable for the particular project, a detailed test plan needs to be created (Section 11.4.1).

The selection of test levels typically is a trade-off between test depth and test efficiency. A risk-based approach can be chosen to balance both aspects. In order to facilitate efficient testing, it is recommended to reuse artifacts and data from previous levels and to clearly define the objective for each test level.

11.4.3 Selection of Test Objects

Test levels could be divided into two categories:

- Test levels, such as embedded system testing, focusing on one particular test object (e.g., the embedded system).
- Test levels, such as model testing and software testing, covering a number of components as potential test objects (e.g., all components of the model/software).

For the latter, a choice of model or software components to be tested at this level is usually necessary due to the limitation of development resources and time. As an example, not all subsystems of a Simulink model might be subjected to a model component test. Selecting model or software components (in the following named artifacts) to be tested can be based on one or more of the following selection criteria:

- In *function-oriented selection*, artifacts that realize a distinguished function are selected as test objects. Note that the chosen artifacts must be separable from the overall system and thus also be testable. Those artifacts that are directly addressed by a particular requirement should also be selected as test objects to be able to check the fulfillment of that requirement. Artifacts containing complex algorithms should be tested separately, if the complexity of the calculation cannot be sufficiently taken into account by other tests. Here, an independent test should be carried out on the lowest possible integration level.

- In *structure-based selection*, the selection of test objects can be based on the structure or complexity of artifacts and/or on their intended deployment. To guide the selection, different size or complexity metrics can be applied to the possible artifacts. Metrics could include information on interface size, lines of code, nesting depth, etc. Furthermore, the test object selection may be based on the future hardware allocation or on the task structure.
- In *personnel-oriented selection*, the selection of test objects resembles the division of work between the individuals involved in the development of the artifacts. In addition, the developer's expertise can also provide information on how detailed such tests need to be.
- In *risk-based selection*, the artifacts to be tested are identified on the basis of a risk assessment (risk-based testing). The amount of test resources corresponds to the criticality of the artifact. Risk-based test selection can be also applied to determine a suitable order of test objects in case of limited test resources.
- In *resource-oriented selection*, the selection and intensity of the tests correspond to the resources available for the test. While this criterion is by all means relevant from the project management's point of view, it should not be the primary decision-making basis for test planning. Possible delays in development could lead to entire tests being discontinued or disproportionately reduced for reasons of time and cost reduction.
- In *phase or maturity-level-oriented selection*, the test object selection is related to the current development phase or the maturity level of individual development artifacts, respectively. This ensures that only those artifacts are thoroughly tested that have reached a more or less stable state of development and are relevant in the current development phase (e.g., for the development of a B-sample control unit). However, since this criterion only takes into consideration a present view of the development phase, the global selection of test objects must be done independently of this estimation. Otherwise, test gaps may result.

11.4.4 Integration Strategies

In accordance with the selection of individual test objects, a proper "integration strategy" should be established for all applicable integration test levels. The integration strategy outlines how the single subcomponents are to be integrated step by step into the overall system and how they are to be tested during integration testing. Integration testing (Sections 11.3.1.1 and 11.3.2.1) typically resembles the integration strategy and order.

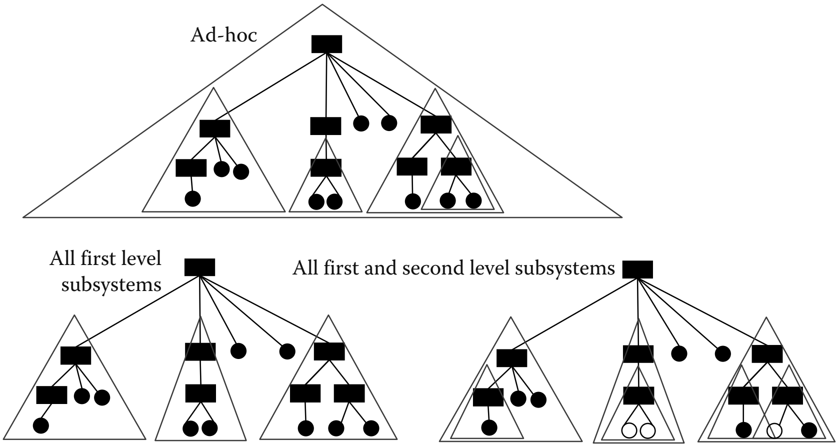

Ad-hoc
All first level subsystems
All first and second level subsystems

FIGURE 11.24 Common integration test strategies.

Not having a well-defined integration strategy regularly comes along with an "ad hoc selection of test objects" (Figure 11.24, upper part) resulting in test gaps and/or redundancies.

Testing literature proposes various integration strategies, such as "bottom-up, top-down, middle-out, or big-bang integration" [Mye79]. However, the development paradigm chosen restricts the number of applicable integration strategies.

Small and medium size model-based development projects, for example, commonly use a big-bang approach. If aggregated components can only be tested in a closed-loop manner, then big-bang integration is also a good choice. Other integration strategies would produce a partly integrated model or software that does not match the interfaces of the plant model. Creating stubs for the missing components is probably not worth the effort.

In the scope of model-based development a big-bang integration of all first (upper) level model components (subsystems) in combination with subjecting all first level or all first and second level components to open-loop model testing (Figure 11.24 lower part) has been proven successful. Following the integration an open- or closed-loop model testing of the integrated model (i.e., model system testing) should be executed.

Beside a complete integration sequence, which is most advantageous from the testing perspective, there are normally other constraints such as the sequence of the completion and delivery of artifacts that have to be taken into account. The defined integration strategy should also consider these constraints.

The integration strategy should be documented as part of the detailed test plans for those test levels that comprise the integration of components, for example, MIL #1, #2, SIL, and PIL.

11.4.5 Test Environments

Apart from the various test objects, different test "execution platforms" are particularly important in the development of automotive systems. Relevant execution platforms in this context include the modeling and simulation environment used (if applicable), the host computer or the target processor, and the complete ECU. Depending on the development paradigm and the type of the application some of these platforms might not be available. In some cases, it makes more sense to choose just one of these platforms as execution environment. Target-hardware-independent execution platforms particularly play an important role in the early stages of developing new systems, as the target hardware often is not available for tests at this time.

The *test environment* includes the definition of necessary real or simulated environment elements, in addition to the platform on which the test object is to be executed. The required effort to provide particular environment elements depends on the type of the application to be developed. In the development of vehicle dynamics applications, for example, effects of road conditions, the vehicle system's hardware, and the behavior of the driver often play a decisive role in testing. Proper test planning in this context includes to ensure the availability of appropriate parts of the plant and environment models and real hardware components.

Considerations related to the test environments should be documented in the detailed test plans.

11.5 Summary

In automotive software engineering, dynamic testing forms the core element of analytical quality assurance. Prerequisites for a thorough and systematic test are a careful test planning and a structured testing process covering test levels and testing activities. For testing embedded automotive software, a variety of testing techniques has emerged that has significant differences to comparable general-purpose testing techniques developed for nonembedded software.

Among the core test activities, test design, that is, the selection of suitable test scenarios, is the most important one since it determines the extent and the quality of the test. To achieve the given test objectives, a single test design technique is not sufficient. Rather a number of complementing test design techniques need to be blended systematically into an overall test strategy. Test strategies for embedded automotive software typically differ depending on the development paradigm used. Since the different executable models could be exploited as additional, comprehensive sources of information for testing, new possibilities for the test arise in the context of model-based development. The broader spectrum of test possibilities allows to establish more flexible test approaches. In particular, inexpensive model tests can be carried out first. Then, comparative back-to-back tests between consecutive executable artifacts can be performed to verify the transition from one artifact to its successor.

The test strategy needs to be determined upfront and to be laid in a master test plan that forms the basis for the detailed test plans of the individual test levels.

Beside the exceptional significance of dynamic testing for the quality assurance of in-vehicle software it is indispensable to integrate testing with other verification and validation techniques. A combination of dynamic testing with automated static analyses and manual reviews has proven successful in practice.

References

[Bal98] H. Balzert. Lehrbuch der Software-Technik. Band 2, Spektrum Akademischer Verlag, 1998.

[BCS+03] A. Baresel, M. Conrad, S. Sadeghipour, and J. Wegener. The interplay between model coverage and code coverage. In: *Proceedings of the 11th European International Conference on Software Testing, Analysis and Review (EuroSTAR 03)*, Amsterdam, the Netherlands, 2003.

[Bel98] F. Belli: Methoden und Hilfsmittel für die systematische Prüfung komplexer Software. Informatik-Spektrum 21, S. 337–346, 1998.

[BN03] E. Broekman and E. Notenboom. *Testing Embedded Software*. Addison-Wesley, London, 2003.

[CDF+99] M. Conrad, H. Dörr, I. Fey, and A. Yap. Model based generation and structured representation of test scenarios. In: *Proceedings of the Workshop on Software-Embedded Systems Testing (WSEST '99)*, Gaithersburg, MD, 1999.

[CDS+02] M. Conrad, H. Dörr, I. Stürmer, and A. Schürr. Graph transformations for model-based testing. *Lecture Notes in Informatics, LNI*, P-12:39–50, 2002.

[CFP03] M. Conrad, I. Fey, and H. Pohlheim. Automatisierung der Testauswertung für Steuergerätesoftware. *VDI-Berichte*, 1789:299–315, 2003.

[CH98] M. Conrad and D. Hötzer. Selective integration of formal methods in the development of electronic control units. In: *Proceedings of the Second International Conference on Formal Engineering Methods (ICFEM '98)*, Brisbane, Australia, pp. 144–155, 1998.

[CK06] M. Conrad and A. Krupp. An extension of the classification-tree method for embedded systems for the description of events. In: *Proceedings of the Second ETAPS Workshop on Model Based Testing (MBT 2006)*, Vienna, Austria, pp. 1–9, 2006.

[Con01] M. Conrad. Beschreibung von Testszenarien für Steuergerätesoftware–Vergleichskriterien und deren Anwendung. *VDI-Berichte*, 1646:381–398, 2001.

[Con04a] M. Conrad. Modell-basierter Test eingebetteter Software im Automobil—Auswahl und Beschreibung von Testszenarien. Deutscher Universitätsverlag, 2004.

[Con04b] M. Conrad. A systematic approach to testing automotive control software. In: *Proceedings of the 30th International Congress on Transportation Electronics (Convergence '04)*, Detroit, MI, pp. 297–308, 2004.

[CS03] M. Conrad and E. Sax. Mixed signals. In: *Testing Embedded Software*. Addison-Wesley, London, pp. 229–249, 2003.

[CSW06] M. Conrad, S. Sadeghipour, and H.-W. Wiesbrock. Automatic evaluation of ECU software tests. In Proc. SAE World Congress 2005, *Journal of Passenger Cars*—Mechanical Systems, SAE International, Detroit, MI, March 2006.

[DSS+01] M. Dornseiff, M. Stahl, M. Sieger, and E. Sax. Durchgängige Testmethoden für komplexe Steuerungssysteme—Optimierung der Prüftiefe durch effiziente Testprozesse. *VDI-Berichte*, 1646:347–366, 2001.

[GG93] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3:63–82, 1993.

[Gri88] K. Grimm. Methdoden und Verfahren zum systematischen Testen von Software. *Automatisierungstechnische Praxis atp*, 30(6):271–280, 1988.

[Gri95] K. Grimm. Systematisches Testen von Software—Eine neue Methode und eine effektive Teststrategie. Dissertation, GMD-Bericht Nr. 251, Oldenbourg, 1995.

[GS05] M. Grochtmann and L. Schmuhl. Systemverhaltensmodelle zur Spezifikation bei der modellbasierten Entwicklung von eingebetteter Software im Automobil. In: *Proceedings of the Modellbasierte Entwicklung eingebetteter Systeme (MBEES'05)*, Dagstuhl, Germany, 2005, pp. 37–42.

[Höt97] D. Hötzer. Schaltstrategieentwurf mit Statemate unter Einbindung kontinuierlicher Modelle zur Softwareverifikation. In: *Proceedings of the Fifth Statemate Anwenderforum*, München, Germany, 1997.

[HT01] R. Helldörfer and U. Teubert. Automated software verification at TEMIC. *dSPACE News* 1/2001.

[IEEE 610.12] IEEE Std. 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. Institute of Electrical and Electronics Engineers, Inc., New York, 1990.

[ISO 15497] ISO/TR 15497:2000. *Road Vehicles—Development Guidelines for Vehicle Based Software*. International Organization for Standardization, Geneva, Switzerland, 2000.

[LBE+04] K. Lamberg, M. Beine, M. Eschmann, R. Otterbach, M. Conard, and I. Fey. Model-based testing of embedded automotive software using MTest. In: Proc. SAE World Congress 2004, Detroit, MI, March 2004.

[Lig90] P. Liggesmeyer. Modultest und Modulverifikation: State of the Art. Angewandte Informatik Band Vol. 4, BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Lig92] P. Liggesmeyer. Testen, Analysieren und Verifizieren von Software—eine klassifizierende Übersicht der Verfahren. In: P. Liggesmeyer et al. (HrsgEds.): Testen, Analysieren und Verifizieren von Software. Reihe Informatik aktuell, S. 1–25, Springer, Berlin, 1992.

[Mye79] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, 1979.

[Rau02] A. Rau. Verwendung von Zusicherungen in einem modellbasierten Entwicklungsprozess. *Informationstechnik und Technische Informatik it + ti*, 44(3):137–144, 2002.

[RCK+00] A. Rau, M. Conrad, H. Keller, I. Fey, and C. Dziobek. Integrated model-based software development and testing with CSD and MTest. In: *Proceedings of the International Automotive Conference (IAC)*, Stuttgart, Germany, 2000.

[RWS+01] C. Ritter, J. Willibald, E. Sax, and K. D. Müller-Glaser. Entwurfsbegleitender Test für die modellbasierte Entwicklung eingebetteter Systeme. 13. In: *Workshop Testmethods and Reliability of Circuits and Systems*, Miesbach, Germany, 2001.

[Sim97] D. Simmes. Entwicklungsbegleitender Systemtest für elektronische Fahrzeugsteuergeräte. Herbert Utz Verlag Wissenschaft, 1997.

[SL] The MathWorks, Inc.: Simulink®—Simulation and Model-Based Design. www.mathworks.com/products/simulink (2007/01/10).

[SF] The MathWorks, Inc.: Stateflow®—Design and simulate state machines and control logic. www.mathworks.com/products/stateflow (2007/01/10).

[SWM02] E. Sax, J. Willibald, and K. D. Müller-Glaser. Seamless testing of embedded control systems. In: *Third IEEE Latin-American Test Workshop*, Montevideo, Uruguay, 2002.

[SZ05] J. Schäuffele and T. Zurawka. *Automotive Software Engineering—Principles, Processes, Methods, and Tools*. SAE International, 2005.

[TAV96] Fachgruppe 2.1.7 Test, Analyse und Verifikation von Software (TAV) der Gesellschaft für Informatik (GI): Begriffsdefinitionen im Testbereich.Working Draft, 1996.

[UPL06] M. Utting, A. Pretschner, and F. Legeard. A taxonomy of model-based testing. Technical report 04/2006, Department of Computer Science, The Universiy of Waikato, New Zealand, 2006.

[Wat82] M. L. Watkins. A technique for testing command and control software. *Communications of the ACM*, 25(4):228–232, 1982.

[WCF+02] H.-W. Wiesbrock, M. Conrad, I. Fey, and H. Pohlheim. Neue automatisierte Auswerteverfahren für Regressions- und Back-to-back-Tests eingebetteter Regelsysteme. *Softwaretechnik-Trends*, 22(3):22–27, 2002.

[Weg01] J. Wegener. Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen. Shaker Verlag, 2001.

[WGP02] S. Weber, A. Graf, and D. Peters. Automated integration testing of powertrain software with LabCar ST. In: *Proceedings of the ETAS Competence Exchange Symposium 2002*, Stuttgart, Germany, 2002.

12

Testing and Monitoring of FlexRay-Based Applications

Roman Pallierer
Elektrobit Corporation

Thomas M. Galla
Elektrobit Corporation

12.1 Introduction to FlexRay-Based Applications

FlexRay has been developed for future in-car control applications demanding high data rates, deterministic behavior, and able to support fault tolerance. Application domains of FlexRay-based systems include power train, chassis, and body control. Furthermore, FlexRay is considered as a backbone network interconnecting several main electronic control units (ECUs). This chapter focuses on the testing and monitoring concepts for such applications interconnected via FlexRay.

12.1.1 System Architecture

When describing the system architecture, a distinction between the hardware and the software aspects of the system architecture must be made. The hardware architecture shown in this chapter reflects the state of the art in today's automotive systems. The software architecture presented in this chapter conforms to the AUTomotive Open

System ARchitecture (AUTOSAR) standard [1] (see Chapter 2). However, some simplifications are made for the sake of brevity, clarity, and simplicity (i.e., unnecessary details are sometimes omitted).

12.1.1.1 Hardware Architecture

The hardware architecture of automotive systems can be viewed at different levels of abstraction.

The "system level" is at the highest level of abstraction. An automotive system consists of a number of networks interconnected via gateways (Figure 12.1). In general, these networks correspond to the different functional domains that can be found in today's cars (i.e., chassis domain, power train domain, body domain).

The networks themselves comprise a number of ECUs that are interconnected via a communication media (see zoom-in on network A and D in Figure 12.1). The physical topology used for the interconnection is basically arbitrary; however, bus, star, and ring topologies are the most common in today's cars. This level, named "network level," represents the medium level of abstraction.

Note that, conceptually speaking, a gateway is a special ECU that is actually a member of all networks that are interconnected by this gateway.

At the lowest level of abstraction there is the "ECU level" (Figure 12.2). Here, the major parts of an ECU are of interest. An ECU is comprised of one or more microcontroller units (MCUs) as well as one or more communication controllers (CCs). In most cases, only one MCU and one CC are used to build up an ECU.

In order to be able to control the physical processes in a car (e.g., control the injection pump of an engine) the ECU's MCU is connected to actuators via the MCU

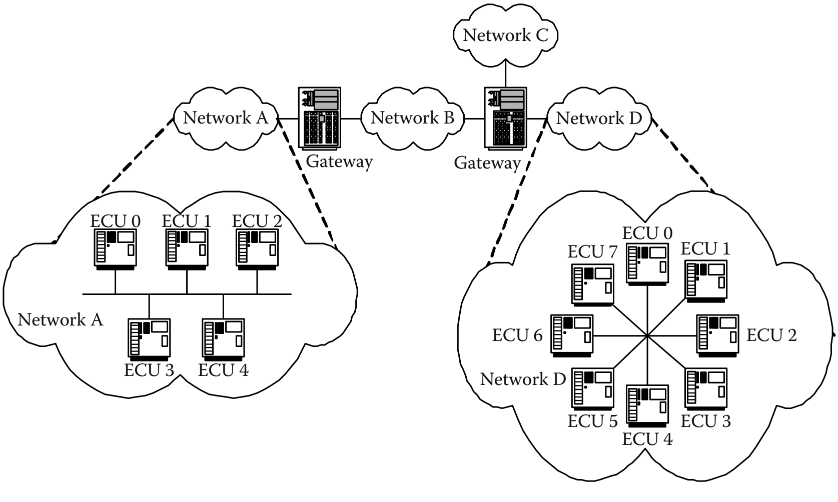


FIGURE 12.1 Hardware architecture—system and network level.

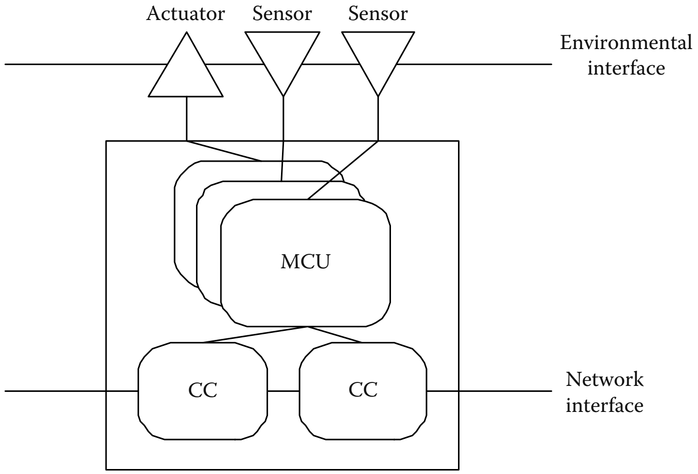


FIGURE 12.2 Hardware architecture—ECU level.

analog or digital "output" ports. To provide means to obtain environmental information, sensors are connected to the MCU analog or digital "input" ports. We call this interface the ECU's environmental interface.

The CC(s) facilitate(s) the physical connectivity of the ECU to the respective network(s). We call this interface the ECU's network interface. The number of CCs hosted by gateway ECUs thus usually equals the number of networks interconnected by the respective gateway.

12.1.1.2 Software Architecture

The AUTOSAR software architecture makes a rather strict distinction between application software and basic or system software. While the basic (or system) software provides functionality like communication protocol stacks for automotive communication protocols (e.g., FlexRay), an operating system and diagnostic modules, the application software is comprised of all application-specific software items (i.e., control loops, interaction with sensors and actuators, etc.). This way, the basic or system software provides the foundation upon which the application software is built.

12.1.1.2.1 Application Software

Application software in AUTOSAR consists of application software components, which are ECU and location independent and sensor–actuator components that are dependent on ECU hardware and therefore location dependent. Whereas instances of application software components can easily be deployed to and relocated among different ECUs, instances of sensor–actuator components must be deployed to a specific ECU for performance/efficiency reasons. Deploying multiple instances of the same component to a single ECU is supported by the AUTOSAR component standard. A simple example for the deployment of multiple instances of the same component is an ECU with two redundant sensors. In that scenario, two instances of the respective sensor component would be deployed to the ECU, each instance servicing exactly one of the two sensors.

Application software components as well as sensor–actuator components are interconnected via so-called connectors. These connectors represent the exchange of data, usually called "signals" in automotive domain, among the connected components. The characterics, requirements, and the constraints on such a signal exchange are specified as attributes of the respective connector. Hence, the following classes of characteristics, requirements, and constraints have to be considered.

12.1.1.2.1.1 Timing Characteristics and Requirements This class defines the temporal properties of the signal exchange, namely the properties occurrence, period, latency, and jitter. As far as the "occurrence" of a signal exchange is concerned, a distinction between periodic exchange, sporadic exchange, and aperiodic exchange can be made. While for periodic and sporadic exchanges, constraints on the temporal distance between two consecutive signal exchanges can be specified, no such constraints can be given for aperiodic exchanges. The "period" of the signal exchange is hereby the temporal distance between two consecutive signal exchanges in the case of periodic signals (P in Figure 12.3). In the case of sporadic signals, the period defines the minimum temporal distance between any two consecutive signal exchanges (sometimes also called minimum interarrival time). For aperiodic signals, the period property is not used. The "latency" (L_i in Figure 12.3) of a signal exchange is defined as the temporal distance between the initiation of the signal transmission at the sender (i.e., the point in time the sending application software component calls the sending application programming interface [API] service) and the signal reception at the receiver (i.e., the point in time the received signal is available at the receiving application software component). Given the characteristics of periodic/sporadic signals exchanged through a given network, it is possible to evaluate a priori the worst latency for each signal; for example, in Ref. [2] the authors have shown how to evaluate this worst case for a FlexRay network, while in Chapter 13 in this book, the evaluation method of the same characteristic for a CAN is presented. Properties can be required on the latency of signals such as the "maximum allowable latency" (e.g., L_i has to be always lower than $L_{\max}$ in Figure 12.3) or an imposed mean value M (e.g., $(\sum_i L_i/i)$ has to be equal to M in Figure 12.3). The deviation of the

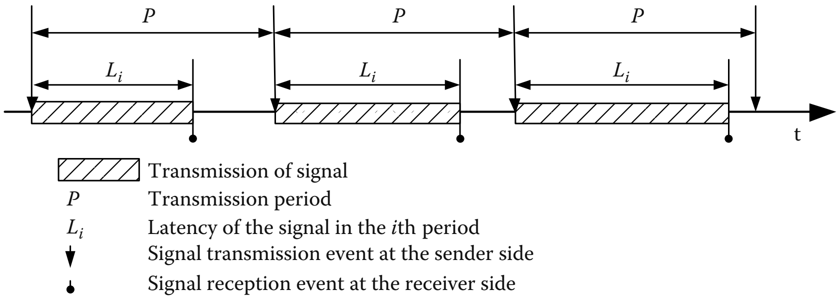

P
P
P
L_i
L_i
L_i
t
▨ Transmission of signal
P Transmission period
L_i Latency of the signal in the ith period
↓ Signal transmission event at the sender side
↓ Signal reception event at the receiver side

FIGURE 12.3 Characteristics for a periodic signal exchange.

actual observed latency of the exchange of a specific signal from the mean latency is termed "jitter." Since minimizing the jitter is of utmost importance to ensure high quality distributed control loops, the "maximum allowable jitter" is another important attribute of the connector (e.g., $\mathrm{abs}(L_i - M)$ has to be less than a given value $J_{\max}$ in Figure 12.3). Note that the transmission guarantee requirements (e.g., guaranteed vs. best effort transmission) can easily be expressed by means of the mean latency and the maximum allowable jitter parameter. In particular, setting the required mean latency of a connector, for example to a value different from infinity and requiring that the maximum jitter is smaller than a defined value, formulate a requirement for a guaranteed transmission with bounded jitter. Setting the required mean latency to infinity, however, and requiring a maximum jitter that is smaller than a defined value, formulate a requirement for a nonguaranteed transmission, which, in the case where the transmission takes place, has a bounded jitter.

12.1.1.2.2 Fault-Tolerance Requirements

This class defines the fault-tolerance properties of the signal exchange, namely the properties' redundancy type, redundancy degree, and additional parameters for a certain redundancy type. As far as the "redundancy type" is concerned, a distinction between spatial redundancy (i.e., signal exchange via multiple physical communication channels) and temporal redundancy (i.e., performing the signal exchange multiple times with the same signal value within a given interval) can be made. The number of different physical communication channels or the number of time-redundant signal exchanges within a specific interval is defined by the property "redundancy degree." Since both types of redundancy (namely spatial and temporal) can be combined for a single signal exchange, a separate instance of the redundancy property is required for spatial and temporal redundancy. For temporal redundancy, an additional attribute is required to specify the minimum temporal distance between two consecutive replicas of a signal exchange. The rationale behind this attribute is the requirement that, for example, disturbance bursts with a maximum duration of ε have to be tolerated. In this case, the minimum temporal distance between two consecutive replicas of the signal exchange has to be larger than the maximum disturbance burst duration ε in order for this kind of burst to be tolerated. For further information on how to determine the distribution of the replica for a time division multiple access (TDMA)-based protocol, you can refer to Ref. [3]. Explicitly specifying the redundancy type is not required, since this information is implicitly defined via the spatial and the temporal redundancy degree.

12.1.1.2.3 Basic or System Software

In addition to the application software components, AUTOSAR also defines a layered architecture of basic (or system) software modules, which provide a basic platform for the execution of the application software components.

The AUTOSAR basic software is horizontally subdivided into different types of services, namely:

- Input/output (I/O) services, which provide standardized access to sensors, actuators, and ECU onboard peripherals
- Memory services, which facilitate the access to internal and external (mainly nonvolatile) memory
- System services, which contain modules like operating system, ECU state management, etc.
- Last, but not least, communication services, which provide a communication stack used for access to the different vehicle networks (i.e., local interconnect network [LIN], controller area network [CAN], and FlexRay)

12.1.1.2.3.1 Communication Services Communication services are a group of modules for vehicle communication (CAN, LIN, and FlexRay). The communication stack built up by the modules of the communication services is depicted in Figure 12.4. The grey boxes indicate communication protocol-specific modules. The "XXX" is a placeholder for the respective communication protocol (i.e., CAN, LIN, and FlexRay). Thus, the AUTOSAR communication services contain communication

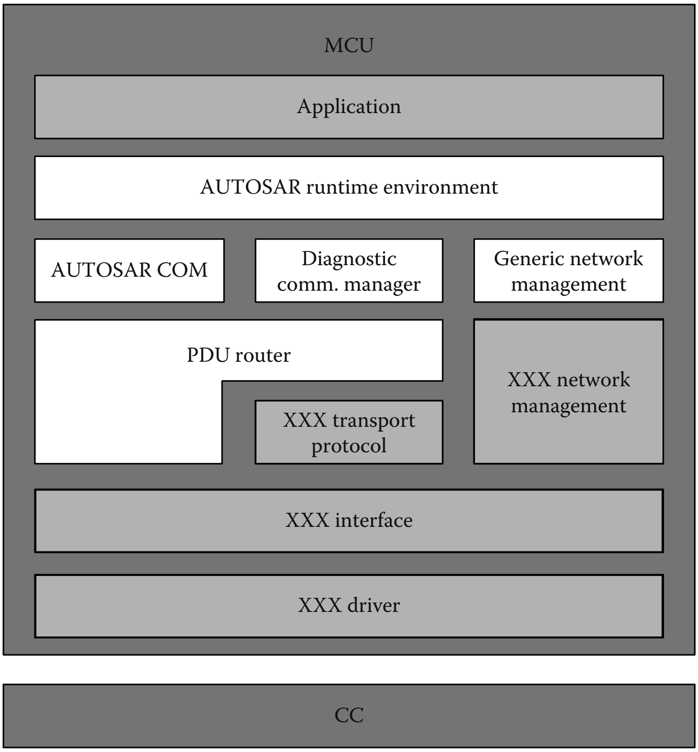


FIGURE 12.4 AUTOSAR communication services.

protocol-specific instances of the transport protocol (TP) and network management (NM).

XXX TP: In AUTOSAR, the TP is used to perform segmentation and reassembly of large protocol data units (PDUs) (called messages) transmitted and received by the diagnostic communication manager (DCM). A dedicated TP is used for each communication protocol (CAN, LIN, and FlexRay). These protocols are rather similar or even compatible (in certain configuration settings) with the ISO TP for CAN [4] specified in ISO/DIS 15765-2.2.

PDU router: The PDU router module provides two major services. On the one hand, it dispatches PDUs received via the underlying interfaces (e.g., FlexRay interface) to the different higher layers (COM, DCM). On the other hand, the PDU router performs gateway functionalities between different communication networks by forwarding PDUs from one interface to another of either the same (e.g., FlexRay → FlexRay) or of a different type (e.g., CAN → FlexRay).

COM: The COM module provides signal-based communication to the higher layers runtime environment (RTE). The signal-based communication service of COM can be used for intra-ECU communication as well as for inter-ECU communication. In the former case, COM mainly uses shared memory for this intra-ECU communication, whereas for the latter case, COM packs multiple signals into a PDU at the sender side and forwards this PDU to the PDU router in order to issue the PDU's transmission via the respective interface. On the receiver side, COM obtains a PDU from the PDU router, extracts the signals contained in the PDU, and forwards the extracted signals to the higher software layers.

DCM: The DCM provides services that allow a tester device to control diagnostic functions in an ECU via the communication network (i.e., CAN, LIN, and FlexRay). The DCM supports KWP2000 [5], standardized in ISO/DIS 14230-3, and the unified diagnostic services (UDS) protocol [6], standardized in ISO/DIS 14229-1.

NM: NM provides means for the coordinated transition of the ECUs in a network into and out of a low-power (or even power down) sleep mode. AUTOSAR NM is divided into two modules: a communication-protocol-independent module (generic NM) and a communication-protocol-dependent module (CAN NM, LIN NM, and FlexRay NM).

XXX interface: The interface module is protocol specific, meaning that dedicated interfaces for the different communication protocols (i.e., FlexRay, CAN, and LIN) do exist. Based on the frame-based services provided by the respective drivers (see below), the interface modules facilitate the sending and receiving of PDUs, where multiple PDUs can be packed into a single frame at the sending ECU and have to be extracted again at the receiving ECU. In FlexRay, the point in time when this packing and extracting of PDU takes place, as well as the point in time when the frames containing the packed PDUs are handed over to the respective driver for transmission or retrieved from the driver upon reception, are governed by the temporal scheduling of so-called communication jobs of the FlexRay Interface. Thus, each communication job can consist of one or more "communication operations," each of these communication operations handling exactly one communication frame (including the PDUs contained in this frame).

XXX driver: Just like the interface module, the driver module is protocol specific as well. The driver module provides the basis for the interface module by facilitating the transmission and the reception of frames via the respective CC.

RTE: The AUTOSAR RTE provides the interface between application software components and the basic software modules as well as the infrastructure services that enable communication to occur between application software components.

Application layer: Actually, this layer is not part of the AUTOSAR basic software modules' layered architecture, since this layer contains the AUTOSAR application software components described in Section 12.4.

When looking at FlexRay driver, FlexRay interface, FlexRay transport protocol, and COM, communication at different levels of abstraction and granularity is facilitated, namely frame level, PDU level, message level, and signal level. Note that all of the previously listed requirements can be applied to any of these different levels of abstraction.

12.1.2 FlexRay Protocol

In 2000, BMW, DaimlerChrysler, Philips, and Freescale (Motorola) founded the FlexRay consortium [7,8] with the objective to develop a new communication protocol for high-speed control applications in vehicles for increasing safety, reliability, and comfort. Since then, the consortium has grown to more than 100 members, including some of the automotive industry's largest and most influential players, such as General Motors, Ford, and Bosch among others. In 2006, the BMW Group implemented the first FlexRay-based series application in the X5 family [9], demonstrating the performance of this new communication technology on the road.

The FlexRay protocol provides fast, deterministic, and fault-tolerant communication to overcome the performance limits of previously established protocols in the automotive domain, for example, CAN. Therefore, FlexRay supports two communication channels, each operating at a data rate of up to 10 Mbps. The FlexRay communication scheme includes a static segment and a dynamic segment. Data transmission in the static segment is fully deterministic with guaranteed frame latency and jitter, while the dynamic segment provides a flexible bandwidth allocation for asynchronous data transmission. For the deployment of the FlexRay protocol, all parameters of the communication scheme, such as the length and properties of the static and dynamic segment, have to be configured statically. These parameters depend highly on the requirements of the application.

Testing and monitoring approaches have to take into consideration the detailed configuration parameters of FlexRay. Furthermore, the deterministic timing of the static and dynamic segments can be used to establish an efficient simulation environment.

For a detailed description of the static and dynamic segments, and the timing in FlexRay please refer to Chapter 5.

12.2 Objectives for Testing and Monitoring

Automotive systems often have to meet dependability requirements due to the inherent safety-critical nature of these kinds of systems (especially as far as the chassis domain is concerned). According to Laprie et al. [10], testing is one means to establish the desired amount of dependability. Through testing, the following goals can be achieved:

Fault removal: In the development phase of a system, design faults can be detected by means of testing and can be removed from the system, thus resulting in a higher dependability of the system.

Fault forecasting: When exposed to realistic load scenarios and when supplied with input that is close to real life, the frequency and the severity of faults can be assessed prior to system deployment (i.e., prior to starting car production). Based on this data, forecasts can be made regarding faults occurring in the field.

Basically, a distinction between static testing and dynamic testing can be made. Static testing comprises practices to verify the system without actual execution. Practices like static analysis (e.g., inspections, walk-throughs, data flow analysis, complexity analysis, static source code checks by compilers or dedicated source code checkers) or proving theorems by means of prover engines fall into this category. Since static testing of FlexRay-based systems is not fundamentally different from static testing of non-FlexRay-based system, we will not address the practices of static testing any further.

In dynamic testing, the system is exercised with a defined set of stimuli (the so-called test vectors) in order to judge—based on the responses of the system to these test stimuli—whether the system behaves according to its specification or whether the systems' responses deviate from this specification. Such a deviation from the system's specification is termed a failure of the system.

When conducting dynamic tests on a system, however, it is important to have proper means to monitor and record the response of the device under test (DUT) to the test stimuli. In networked systems, the test stimuli can, to a large degree, be provided via the communication media. Similarly, the responses of the DUT are to a large degree visible on the communication media as well. Therefore, to properly test (parts of) networked systems, some kind of monitoring device to record the network traffic as well as some device capable of providing the proper stimuli via the network are required.

12.2.1 Criteria to Test and Monitor

When testing a system (or a part of a system), the main interest lies in finding out whether the system behaves according to its specification or whether the observed behavior of the system deviates from the system's specification. Such a deviation can take place either in the time domain, value domain, or code domain.

12.2.1.1 Deviations in the Time Domain

When looking for deviations in the temporal domain, all timing-related requirements listed in Section 12.1.1.2.1 have to be taken into consideration and have to be applied at the different levels of granularity (i.e., frame level, PDU level, message level, and signal level).

As far as the period requirement of a connector is concerned, tests have to be conducted to validate that, for periodic or sporadic information exchanges, the observed period matches the required period. In FlexRay, for frames scheduled in the static segment, this period is guaranteed by the FlexRay protocol in fault-free scenarios. For frames scheduled in the guaranteed part of the dynamic segment, the observed period may deviate by a maximum of almost the length of the dynamic segment (Figure 12.5).

The upper part of Figure 12.5 illustrates the case, where no other frames are sent into the minislots prior to frame D (for which the period is observed), resulting in a period of a whole FlexRay communication cycle. In the lower part of Figure 12.5 a scenario is depicted where the minislots preceding the minislot for the transmission of frame D are occupied (and thus stretched) causing the transmission of frame D to be shifted to the end of the communication cycle, resulting in an observed period of one FlexRay communication cycle plus (in a worst-case scenario with long frames and short minislots) almost the length of the dynamic segment.

For the best effort part of the dynamic segment, no such upper bound on the possible deviation can be given since indefinite postponement of the transmission of frames scheduled in the best effort part of the dynamic segment might take place. For data entities different from frames (which require the involvement of higher software layers) like PDUs, messages, or signals, the period is governed by the temporal schedule of the FlexRay interface's communication operations as well. Figure 12.6 illustrates this

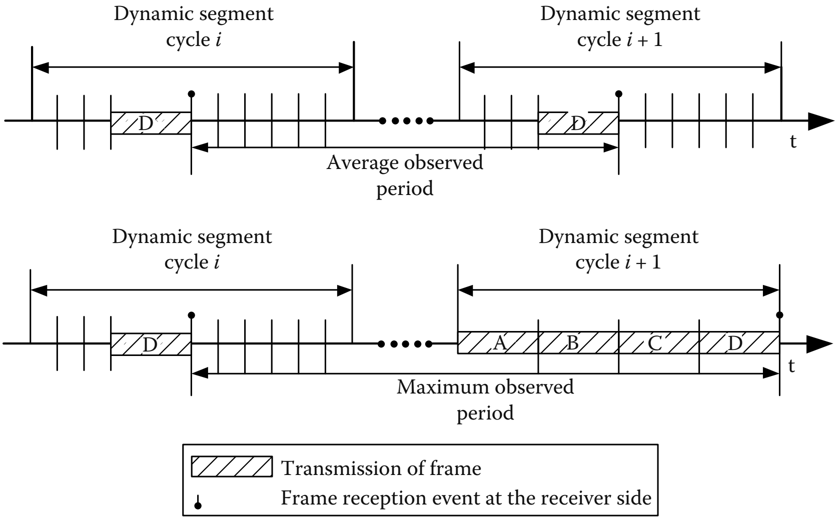


FIGURE 12.5 Observed period in guaranteed part of dynamic segment.

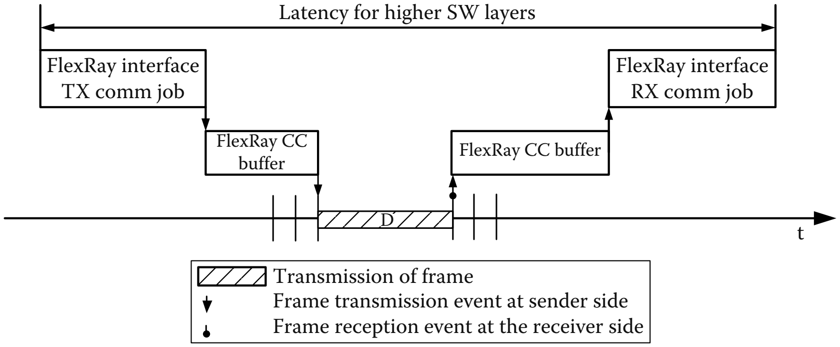

Latency for higher SW layers
FlexRay interface TX comm job
FlexRay CC buffer
FlexRay CC buffer
FlexRay interface RX comm job
D
t
▨ Transmission of frame
Frame transmission event at sender side
Frame reception event at the receiver side

FIGURE 12.6 Impact of FlexRay interface communication operations.

impact of the temporal schedule of the FlexRay interface's communication jobs on the actual latency between the send request (issued by the layer on top of the FlexRay interface) and the actual send event on the communication media (and vice versa on the recipient side).

For the mean latency requirement, the values of the latencies observed by the receiver have to be measured and the mean value has to be computed. Again, for frames transmitted in the static segment of FlexRay, the FlexRay protocol itself ensures a constant latency of one TDMA slot due to the static schedule. In the guaranteed part of the static segment, the mean latency will be in the granularity of the length of the dynamic segment (Figure 12.7), whereas in the best effort part of the dynamic segment, the mean latency can even be unbounded if the network load is high.

For data entities different from frames (which require the involvement of higher software layers) like PDUs, messages, or signals, similar to the observed period, the mean latency is governed by the temporal schedule of the FlexRay interface's communication jobs as well.

As far as the maximum latency jitter requirement is concerned, the FlexRay protocol causes frames scheduled in the static part to show a maximum jitter in the granularity of a single macrotick. Frames scheduled in the guaranteed part of the dynamic segment might exhibit a latency jitter of up to the length of the dynamic segment (i.e., the difference between L^{dyn}_{max} and L^{dyn}_{min} in Figure 12.7), whereas frames scheduled in

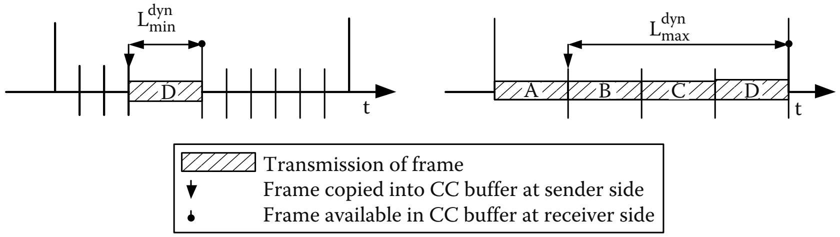

L^{dyn}_{min}
D
t
L^{dyn}_{max}
A
B
C
D
t
▨ Transmission of frame
Frame copied into CC buffer at sender side
Frame available in CC buffer at receiver side

FIGURE 12.7 Latency and latency jitter in dynamic segment.

the best effort part of the dynamic segment may be postponed indefinitely, causing an infinite jitter.

Again, for data entities different from frames (which require the involvement of higher software layers) like PDUs, messages, or signals, the maximum latency jitter is governed by the temporal schedule of the FlexRay interface's communication operations as well.

For testing the temporal redundancy degree requirement, the number of observed temporal replicas of a information exchange has to be counted and compared against the required temporal redundancy degree. Since neither the FlexRay protocol nor the AUTOSAR basic software provides any inherent support for temporal redundancy, the proper handling of temporal redundancy is a matter for the application software.

Meeting the minimum temporal distance requirement between two consecutive replicas of the information exchange is partly supported by the FlexRay protocol. By scheduling the replicas in proper TDMA slots (with a sufficient temporal distance between the slots), the temporal distance requirement is enforced by the FlexRay protocol when using the static segment. For the dynamic segment a worst-case calculation can be made (assuming that all minislots between the minislots of the consecutive replicas are unoccupied) in order to have the FlexRay protocol ensure this requirement. Since this approach, however, is based on a rather pessimistic assumption, the observed temporal distance will mostly be way larger than the minimum temporal distance.

12.2.1.2 Deviations in the Value Domain

Two main kinds of deviations in the value domain can be observed. First, the information content is invalid, since a protecting checksum (for FlexRay frames, e.g., a frame cyclic redundancy check [CRC]) indicates that the information has been mutilated.

Secondly, information content that differs from a known content leads to the conclusion that there is a deviation in the value domain. Nevertheless, in order to come to this conclusion, knowledge about the correct information content is required. For information entities of limited range (e.g., enumeration values), exact knowledge about the correct information is often available (e.g., because the tester knows the exact position of the ignition key). For information entities of a rather large range (e.g., for signal values of 32 bits), however, in most cases only a validity interval is available. In that case, the observed information content can only be validated against this validity interval.

12.2.1.3 Deviations in the Code Domain

When looking at deviations in the code domain, the following deviations have to be considered.

- The bit encoding on the physical layer differs from the specification. This deviation is mostly caused by faulty transceivers and/or encoding units in the FlexRay controller.
- The observed frame format on the data link layer differs from the frame format defined in the FlexRay specification. Such a deviation can be

caused by a faulty transmitter unit of the FlexRay CC or a faulty star coupler.

- And lastly, the observed signal packing (i.e., the ways signals are packed into frames) differs from the specified signal layout in the frame. Such deviations are most probably caused by incorrect configurations of the AUTOSAR COM layer.

12.2.1.4 Other Deviations

For testing the spatial redundancy degree requirement, the number of observed spatial replicas of an information exchange have to be counted and compared against the required spatial redundancy degree. To achieve this, the available channels of the communication system have to be monitored to see if replicas of the information exchange have occurred.

12.2.2 Operational Scenarios for Testing and Monitoring

As far as operational scenarios are concerned, FlexRay-based systems (like any other system) have to be tested in a fault-free case to ensure the system's proper operation when executed in the absence of faults.

As already mentioned in Section 12.1.1, however, FlexRay was developed for deployment in safety-related application areas. Since systems intended for safety-related purposes need to remain functional even in the presence of faults,* the system inherently has to be able to tolerate these faults. Therefore, the testing of safety-related systems has to take place under fault conditions as well, since the previously addressed fault-tolerance requirements on the system mandate that faulty conditions are part of the system's "normal" operational scenario.

In order to be able to test the fault-tolerance properties of the system, the faulty conditions have to be induced intentionally as part of the respective test case. The induction of these faulty conditions is termed fault injection. Depending on whether the faults are injected by means of hardware (e.g., by electromagnetic interference bursts) or by software (e.g., by intentionally flipping single bits in memory) we speak of hardware- or software-implemented fault injection, respectively (Section 12.3.1.2).

12.3 Monitoring and Testing Approaches

In order to achieve the test objectives introduced in the previous section, different monitoring and test approaches are possible. In the following, a basic distinction between software- and hardware-based approaches is made. Software-based validation uses a simulation model of the system to analyze the behavior of the application.

* As long as the number, the frequency of occurrence, the duration, and the nature of these faults is covered by the system's fault hypothesis.

Hardware-based validation includes a hardware setup of the system for the investigation. Both approaches can be used either for the analysis of the total system or for the analysis of parts of the system, that is, one or a number of ECUs. In this chapter, the focus is still on the effects of the software running on the MCU, and therefore, on the application layer and the basic software layers.

12.3.1 Software-Based Validation

Software-based validation provides a powerful means to analyze the application behavior at an early stage of the development.

12.3.1.1 FlexRay Abstraction Levels

Computation effort is a big issue for each simulation. Modeling a complete FlexRay network might become quite complex: distributed ECUs, each including FlexRay controllers and MCUs with basic software layers and the application layers. Furthermore, these networks can be connected with others via gateways (Figure 12.1).

To minimize the complexity of such simulations, only the application is modeled in full detail. The model of the FlexRay controller and network has been significantly simplified, utilizing the deterministic timing behavior of the FlexRay protocol. Therefore, we will introduce so-called abstraction levels on the architecture and timing level.

12.3.1.1.1 Architecture Level

As described in Section 12.1.1.1, the architecture of a FlexRay network consists of ECUs interconnected by a shared communication media. Each ECU includes one or more FlexRay controllers and MCUs. The application layer and the basic software layers run on the MCU. The application layer contains a number of software components that implement the actual application functionality (e.g., anti-blocking system [ABS] calculation routines). The basic software layers provide means and services to transmit and receive data via the FlexRay controllers. The application software components use the services of the basic software layers to communicate with software components running on other ECUs.

Figure 12.8 illustrates this ECU architecture where two interfaces are introduced. First, there is a so-called application interface (AI) between the application layer and the basic software layers, and second, there is a controller–host interface (CHI) between the basic software layers on the MCU and the FlexRay communication controller. These interfaces can be used to facilitate the simulation.

12.3.1.1.1.1 Application Interface At this level, the simulation model contains the full functionality of the software components of the application. The functionality of the basic software layers and the FlexRay controller hardware is emulated providing an AI.

The AI is a signal-based interface that delivers updates of the signals according to the timing of the basic software layers and FlexRay controllers. Emulating this interface in simulation allows considerable simplifications of the simulation model.

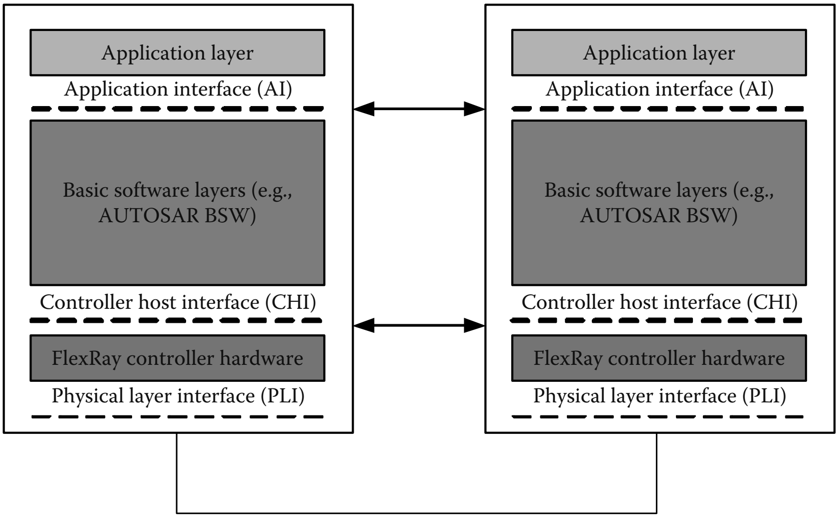

Application layer
Application interface (AI)
Basic software layers (e.g., AUTOSAR BSW)
Controller host interface (CHI)
FlexRay controller hardware
Physical layer interface (PLI)
Application layer
Application interface (AI)
Basic software layers (e.g., AUTOSAR BSW)
Controller host interface (CHI)
FlexRay controller hardware
Physical layer interface (PLI)

FIGURE 12.8 Abstraction level on architecture.

Only the timing aspects of the signal updates have to be considered. Issues like signal-to-frame packing and the details of the FlexRay timing do not need to be modeled.

12.3.1.1.1.2 Controller–Host Interface The full functionality of the software components of the application and the basic software layers is simulated at this level while the functionality of the FlexRay communication controller is emulated, thus providing a CHI abstraction.

The CHI can be modeled as a buffer-based interface that delivers updates of the FlexRay frames according to the FlexRay timing. Emulating this interface in simulation also allows considerable simplifications of the simulation model. Only the timing aspects of the frame updates have to be considered. The FlexRay controller functionality, like clock synchronization and startup etc., does not need to be modeled in detail.

12.3.1.1.1.3 Physical Layer Interface A final interface can also be shown: the physical layer interface (PLI) between the functionality of FlexRay communication controllers and the network physical layer. Simulation at this level basically does not make much sense, since the computational effort required to simulate the correct bit timing (which is required at this level of abstraction) is rather huge. Note that, for hardware-based validation, however (Section 12.3.2), the PLI is an important interface that is subject to faults and thus has to be considered in the validation process.

12.3.1.1.2 Timing Level

Simulation can also be significantly facilitated by choosing an adequate time resolution. Due to the time-driven nature of the FlexRay protocol, the transmission of data

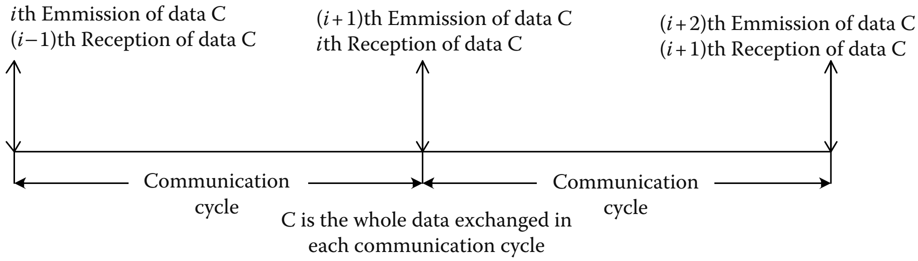

ith Emmission of data C
$(i-1)$th Reception of data C
$(i+1)$th Emmission of data C
ith Reception of data C
$(i+2)$th Emmission of data C
$(i+1)$th Reception of data C
Communication cycle
Communication cycle
C is the whole data exchanged in each communication cycle

FIGURE 12.9 Timing level 1—communication cycle.

is triggered by predefined points in time. This timing hierarchy (Section 12.1.2) can be used to introduce abstraction levels at the timing level.

12.3.1.1.2.1 Timing Level 1: Communication Cycle At this timing level, the updates of frames and the contained signals are performed in the granularity of communication cycles. As shown in Figure 12.9, at the beginning and at the end of each communication cycle, the contents of all signals and frames are updated. The data to be transmitted are written at the beginning of each communication cycle, the data to be received are read at the end of each communication cycle.

This timing level can be optimally combined with the AI level to provide a fast and quite abstract view of the communication timing of signals. It is assumed that transmission latencies introduced by the basic software layers are small enough so that an update of the signal is available at the end of each communication cycle. In combination with the CHI, however, this timing level does not provide the necessary accuracy to analyze the detailed effects of the transmission latency introduced by the basic software layers.

12.3.1.1.2.2 Timing Level 2: Static Slots and Simple Dynamic Segment Arbitration This is a more accurate level where the updates of frames and their contained signals are performed in a more detailed manner. In the static segment, the timing of the static slots is emulated. The data to be transmitted are read at the beginning of each slot, the data received within the slot are provided at the end of the static slot. Figure 12.10 shows the update times for slot 1 of the static segment. The other slots of the static segment are updated in the same way.

For the dynamic segment, only the beginning and the end of the segment are emulated. Data to be transmitted are read at the beginning, data that have been received are provided at the end of the dynamic segment (Figure 12.11).

This timing level can be optimally combined with the CHI level to analyze the effects of the transmission latency introduced by the basic software layers. For example, the AUTOSAR FlexRay interface layer contains communication operations that are scheduled synchronously with the FlexRay communication to read and write to the FlexRay buffers. The configuration of the timing of these communication operations determines the earliest and latest points in time an update of the transmitted

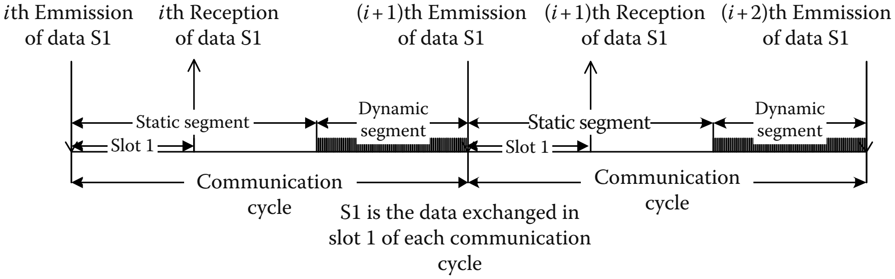

*i*th Emmission of data S1
*i*th Reception of data S1
(*i*+1)th Emmission of data S1
(*i*+1)th Reception of data S1
(*i*+2)th Emmission of data S1
Static segment
Dynamic segment
Slot 1
Static segment
Dynamic segment
Slot 1
Communication cycle
Communication cycle
S1 is the data exchanged in slot 1 of each communication cycle

FIGURE 12.10 Timing level 2—static slot arbitration.

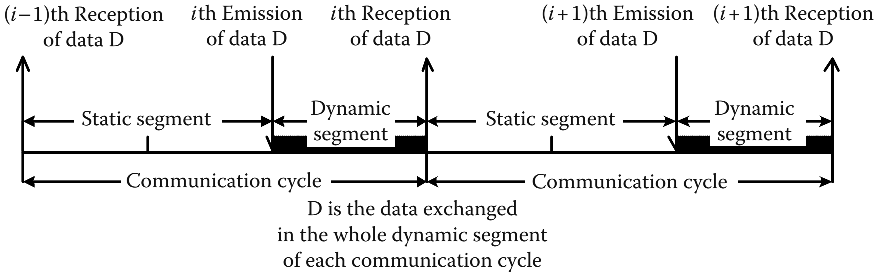

(*i*−1)th Reception of data D
*i*th Emission of data D
*i*th Reception of data D
(*i*+1)th Emission of data D
(*i*+1)th Reception of data D
Static segment
Dynamic segment
Static segment
Dynamic segment
Communication cycle
Communication cycle
D is the data exchanged in the whole dynamic segment of each communication cycle

FIGURE 12.11 Timing level 2—dynamic segment arbitration.

data is available. These effects can be analyzed in combination with the software components of the application layer.

In combination with the AI level, this timing level provides a very accurate update of the frames without taking into consideration any of the latencies introduced by the basic software layers. This may lead to erroneous assumptions about the earliest and latest points in time transmitted data are available.

12.3.1.1.2.3 Timing Level 3: Static Slots and Advanced Dynamic Segment Arbitration This timing level is a refinement of the timing level 2. The sampling points are chosen in the same way as in level 2; however, the behavior of the minislotting media access scheme is emulated in more detail. This priority-based algorithm postpones the transmission of frames scheduled in the best-effort part of the dynamic segment in case of peak load (Section 12.1.2).

12.3.1.1.2.4 More Detailed Approaches Additional simulation approaches include emulating the timing for the dynamic segment in a more detailed way.

12.3.1.2 Fault Injection

Fault injection in the simulation model enables the designer to analyze the dependability characteristics of the system at an early stage of development. The main objectives are (1) fault removal—testing the correct behavior of fault-tolerance mechanisms and (2) fault forecasting—investigating the robustness of the application in fault scenarios.

A simulation model provides a good means to fully control the location and point in time a fault has to be injected. In our simplified model of the FlexRay controller and FlexRay network, we focus on the effects of failures that can be seen on the previously described architectural and timing abstraction levels.

In the simulation model, the AI as well as the CHI, are emulated as shared memories at the architectural level. So, in accordance with the chosen timing abstraction level, the contents of the shared memories can be accessed for fault injection.

12.3.1.2.1 Application Interface

Fault injection on the AI directly provides for influencing and investigating the behavior of the application model. The AI gives access to the services of the basic software layers. In the simulation model, these services are modeled in a very simplified manner, providing signal updates and status information of services such as NM.

Signal updates can be easily influenced in the time and value domains. The time domain depends on the chosen timing abstraction level that corresponds to the update periods of the FlexRay protocol. In contrast to CAN, where the periods of low priority data may significantly jitter even in a fault-free case, FlexRay provides deterministic transmission periods for the static segment and the guaranteed part of the dynamic segment as specified during design time. Even in the instance of an application failing (assuming the basic software layers and the FlexRay controller are working correctly), the update periods of the signals stay constant; however, the signal values may be incorrect or obsolete. This protocol property significantly facilitates the simulation of application faults in the time domain: either a correct signal is available or is not available at the AI within the specified transmission period. Within the best-effort part of the dynamic segment, signal updates may also be lost or postponed in peak-load scenarios, that is, with no specific failures. This behavior can also be easily emulated by mutilating the value of the signal at the AI.

Deviations in the value domain can also be achieved by mutilating the contents of the correct signals provided by the AI. This kind of fault simulation analyzes how the application model reacts if there is wrong input data. Focusing on communication network failures, this method investigates the robustness of the total system when there are inconsistent data transmissions. Due to failures, it might happen that only one subset of ECUs receives signals correctly, while others receive no signal updates.

Other services of the basic software layer, such as NM, can also be influenced by fault injection. The status of the service can be changed at one ECU or at one subset of ECUs to achieve erroneous states and analyze how the application model reacts in those cases.

Timing level 1 is the adequate granularity for these kinds of fault simulation scenarios at the AI. Investigations at finer granularity level (i.e., using a higher timing level) only provide realistic data, when the latency introduced by the basic software layer is also considered. Therefore, the CHI level described below is necessary.

12.3.1.2.2 Controller–Host Interface

Fault injection on the CHI is aimed at analyzing the effects of the basic software layer and the application model. The CHI provides access to the controller status and buffer contents of the FlexRay CC. In the simulation model, the data transmission is emulated by updating the CHIs of all controllers within the specified timing abstraction level. The basic software layers and the application model make use of these values after the update.

At this interface, all failures that might occur in the FlexRay controller or on the communication network can be simulated. These failures include invalid frame receptions, CRC errors, incoming or outgoing link failures, communication channel failures, and more.

The FlexRay specification does not specify the register nor the buffer layout of the CHI of a FlexRay controller. Rather, the services and the contents of the information provided are described. The most important services for the fault simulation are the following.

12.3.1.2.2.1 Clock Synchronization The FlexRay protocol requires that all controllers share a common time base with each other in order to execute, receive, and transmit operations in a time-driven manner. To establish this common time base, a distributed fault-tolerant clock synchronization algorithm [11] is used. Upon power-up, each controller performs a specific startup procedure to establish the common time base. During operation, each controller performs a synchronization algorithm to maintain synchronization with the other controllers. If a controller is not synchronized, it may not fully participate in the data transmission.

The reasons why a controller is not synchronized can be manifold. Examples include no other controllers have powered-up, the FlexRay communication channels are broken, the frequency of the FlexRay controller clock is out of specification, etc.

To analyze the behavior of a FlexRay-based application, we will model only a very abstract view of the clock synchronization algorithm. This abstract view includes two modes: a synchronized and a non-synchronized mode of the FlexRay controller. In the synchronized mode, the controller may correctly receive and transmit frames; in the non-synchronized mode, the controller does not receive correct frames. Furthermore, it is assumed that the controller does not transmit any frames when residing in the non-synchronized mode.

12.3.1.2.2.2 Frame Reception For each frame that is received, a "receive" status is provided in the CHI. This receive status indicates whether the frame has been received correctly or whether any problems have occurred during reception. In our simplified model of the FlexRay controller, the status of the frame reception is mutilated on the CHI so that various failure cases can be emulated. These cases include invalid CRC or check sums, coding violations, clock synchronization problems, etc. These cases can affect one or a certain subset of frames, for example:

- All frames received from a specific node
- All frames received on a specific channel

- All frames received within a specific time period
- Other subsets of frames

This method of selecting certain subsets of affected frames allows for efficiently injecting different kinds of faults. In the simulation model, the behavior of the basic software layers and the application can then be analyzed, for example, whether the failure of one node or communication channel has been correctly detected.

12.3.1.2.2.3 Frame Transmission The success or the failure of a frame transmission results in the value of the receive status in the CHI of all receiving controllers. Thus, mutilating the CHI of all receiving controllers will emulate various frame transmission problems in the simulation model. Furthermore, the CHI of the transmitting controller must be mutilating if it indicates a failed transmit confirmation to the sender.

12.3.1.2.2.4 Choice of Timing Abstraction Levels Fault injection at the CHI helps to analyze the behavior of the basic software layers and the application model. While the reaction of the application model can be investigated at the AI, the behavior of the basic software layer is of specific interest for this type of fault injection. Timing level 1 is not sufficient for these investigations since it does not provide the earliest point in time where frames are available on the CHI. Timing levels 2 and 3, and more detailed approaches, are adequate for these fault simulations at the CHI.

12.3.2 Hardware-Based Validation

In hardware-based validation, the DUT is not simulated in software. In this case, the whole ECU (or even a combination of multiple ECUs) and thus a combination of hardware and its embedded software are put to use in the course of the validation. In order to be able to conduct such a hardware-based validation, the test-bed itself must be hardware-based and the interface between the test-bed and the DUT is a hardware interface. In networked systems, this interface is, on the one hand, the communication media and on the other hand, the I/O interface to the environment (i.e., sensors and actuators).

Based on whether the response of the DUT is or is not fed back into the test vector generation, we can determine either “open-loop” or “closed-loop” approaches.

12.3.2.1 Open-Loop Approach

In open-loop approaches, the responses of the DUT to test stimuli are not fed back into the test stimuli generation to produce new test stimuli. Therefore, the test-bed itself is divided into three main parts, namely a test stimuli generation part, a monitoring part, and a controlling part. While the first is responsible for providing test stimuli to the DUT, the responsibility of the second lies in monitoring the responses to these test stimuli. For coordinating the provisioning of test stimuli and the recording of test responses, the test-bed is comprised of a controlling part as well, which contains a database for retrieving test stimuli and storing test responses for later evaluation.

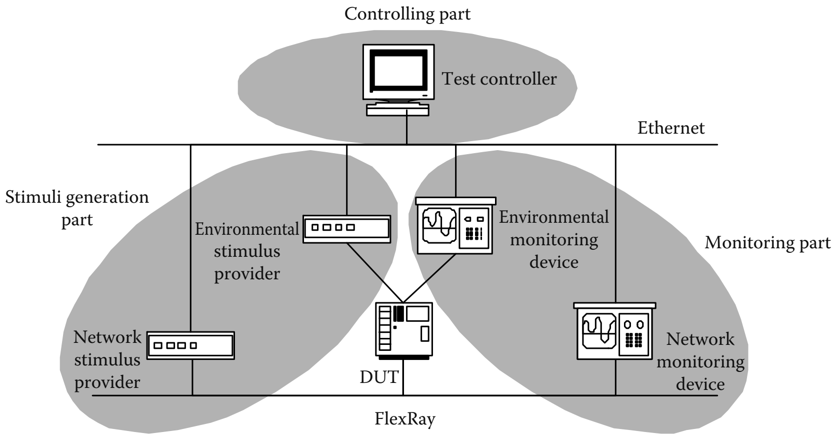

Controlling part
Test controller
Ethernet
Stimuli generation part
Environmental stimulus provider
Environmental monitoring device
Monitoring part
Network stimulus provider
DUT
Network monitoring device
FlexRay

FIGURE 12.12 Test setup for open-loop approach.

In this way, the provided test input as well as the recorded output is generated/obtained at the network interface (network stimulus provider and network monitoring device) and at the environmental interface (environmental stimulus provider and environmental monitoring device) of the DUT. Figure 12.12 illustrates this test setup.

As far as the environmental interface is concerned, FlexRay-based systems are not really different from non-FlexRay-based systems. Thus, we do not further address this interface in the following sections.

12.3.2.1.1 Test and Monitoring Levels

Testing and monitoring in FlexRay-based systems take place in the interfaces introduced in Section 12.3.1.1.1

12.3.2.1.1.1 Physical Layer Interface Testing in fault-free scenarios at the PLI is basically testing the correct operation of the FlexRay CC and the proper setup of the communication media (e.g., proper termination of a bus topology). Given a correctly operating FlexRay CC (including line driver), properly setup communication media, and the absence of faults on the communication media (e.g., no electromagnetic interference [EMI] burst, etc.), no other causes for faults are possible.

With these units of failure in mind, the following faults must be considered during testing (from the monitoring device's perspective):

Faults in the value domain: At the PLI, faults in the value domain mean a deviation of the observed voltage level from the level specified in the FlexRay physical layer specification. This deviation can be caused by improper termination of the communication media, by short-circuiting one or both lines of a FlexRay channel to ground or to supply voltage, or by faults in the star coupler device in star topologies.

Faults in the time domain: One fault in the time domain can be a deviation of the observed bit timing from the specified bit timing, meaning that, for example, the duration of a bit cell is too short or too long. There could also be a deviation of the bit timing in the first derivative; in this case, it means that the edge steepness is either too high or too low. Possible causes for these kinds of faults can be a faulty oscillator in the FlexRay CC, a faulty encoder unit, or an improper configuration of the FlexRay CC.

Faults in the code domain: Deviations from the specified coding are termed coding failures. On the bit level, FlexRay uses not-return-to-zero (NRZ) coding. In order to facilitate bit level synchronization between the transmitter with the receiver(s), each byte is additionally framed with a dedicated start and stop bit (each exhibiting a different logical level), thus enforcing at least one edge per byte. Causes for coding faults are faulty transceivers and/or faulty encoder units in the FlexRay CC.

12.3.2.1.1.2 Controller–Host Interface *Faults in the value domain*: Faults in the value domain at the CHI mean a deviation of frame content from the specified frame content. Subsequently, a distinction has to be established between a frame with incorrect CRC, on the one hand, and a frame with correct CRC and invalid payload, on the other hand. The former case can be caused by a faulty transmitter or a faulty CRC unit of the sending FlexRay CC, or by a faulty MCU, a defect in the application program, or by an improper configuration of the FlexRay CC. In addition to the previous cases, the latter case might be induced by a faulty CHI. Both cases can be generated by faulty communication media as well.

Faults in the time domain: At the CHI, the following faults in the time domain are possible: early frames (i.e., frames are transmitted prior to the specified point in time) and late frames (i.e., frames are transmitted after the specified point in time). Special cases of late frames are the cases where frames are not transmitted at all (omission failure). Possible causes for timing failures are a faulty oscillator in the FlexRay CC, an improper configuration of the FlexRay CC, a faulty star coupler, a faulty communication media, or a faulty MCU or application program.

Faults in the code domain: As far as coding at the CHI is concerned, FlexRay uses a defined frame format consisting of a frame start sequence, a frame header, the frame payload, and a frame trailer containing the frame's CRC. Each of these frame parts has a defined length. Any deviation from this frame format is considered as a coding fault at the CHI. The cause for such a fault is a faulty transmitter unit of the sending FlexRay CC or a faulty star coupler (inducing an oversized truncation of the frame start sequence).

12.3.2.1.1.3 Application Interface *Faults in the value domain*: Faults in the value domain in the AI mean a deviation of signal content from the specified signal content. Hence, a distinction between a signal tagged as invalid and a signal that is tagged as valid but exhibits an incorrect value can be made. Both cases might be caused by a value domain fault on the data link layer, a faulty MCU or application program, or a faulty configuration of the AUTOSAR COM layer.

Faults in the time domain: In the AI, faults in the time domain are either caused by faults in the time domain at the CHI or by late or early activation of communication tasks that are responsible for packing signal to be transmitted into the respective

FlexRay frames. These faults create a deviation in the signal's temporal requirements (Section 12.1.1.2.1).

Faults in the code domain: If we consider coding in the AI, multiple signals are packed into a single frame according to a specified signal layout for each frame. Any deviation from this specified signal layout is interpreted as a coding fault in the AI. Causes for such a coding fault are mostly an improper configuration of the AUTOSAR COM layer or faults in the MCU.

12.3.2.1.2 Testing under Fault Conditions

As stated before, in order to perform a test under fault conditions, fault injection is usually required. In open-loop hardware test setups, the DUT in general cannot be modified. Thus, the DUT must be considered as a black box leaving only the network interface and environmental interface as targets for fault injection. In the following, we focus on the network interface.

With proper hardware modifications, the network stimulus provider (under control of the test controller) can inject all faults described in the previous section into the physical line interface, into the CHI and into the AI as well.

Coding faults at the physical line interface, for example, can be injected by implementing a special encoding unit that provides controlled means to intentionally violate the bit encoding scheme defined in the FlexRay specification. Faults in the value domain in the CHI, for example, can be injected by deliberately producing an incorrect CRC at the network stimulus provider. Value faults at the AI, for example, can be injected by intentionally sending incorrect signal values or by tagging signals intentionally as invalid, while leaving the frame's CRC intact.

12.3.2.2 Closed-Loop Approach

In contrast to open-loop approaches, with closed loop, the responses of the DUT are fed back into the stimuli generation to produce new test stimuli. Thus, the environmental stimuli provider and environmental monitoring device as well as the network stimuli provider and network monitoring device of Figure 12.12 are tightly connected with each other or implemented as one component (Figure 12.13).

This method is often also called "residual bus simulation" (in the German language, "Restbussimulation") or the "hardware in the loop" (HIL) system. HIL systems include a complete model of the other ECUs and a detailed model of the controlled environment, for example, a braking system including the behavior of brakes, the car, and the road. Residual simulation usually includes a simplified model of the other ECUs and no, or only a basic, model of the controlled environment.

The test and monitoring levels for FlexRay-based applications are the same as those presented for open-loop approaches.

12.4 Discussion of Approaches

Having identified several possible testing approaches in the previous section, the question remains: which of these techniques is best suited for the development of

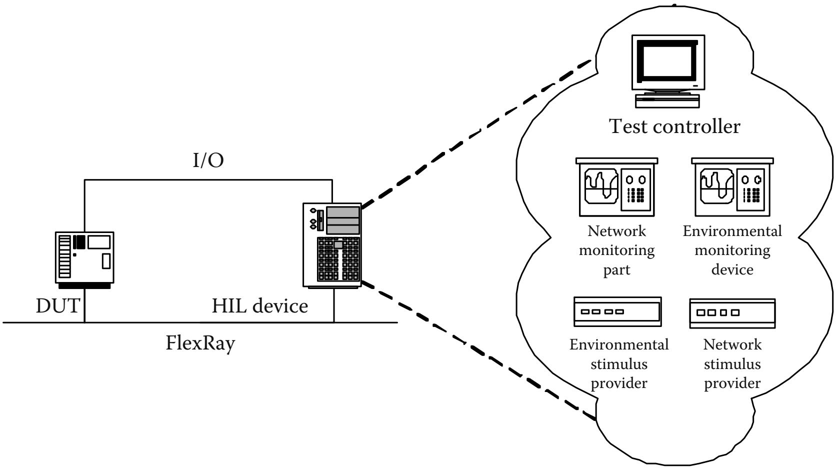

I/O
DUT
HIL device
FlexRay
Test controller
Network monitoring part
Environmental monitoring device
Environmental stimulus provider
Network stimulus provider

FIGURE 12.13 Test setup for closed-loop approach.

FlexRay-based applications and systems? In this section we compare the presented approaches by evaluating them in terms of cost, accuracy, and ease for the testing process.

12.4.1 Software-Based Approaches

Purely software-based approaches, where no target hardware is involved, are flexible and adaptable to the testing purpose, and therefore it is easy to control the execution of tests, to monitor the test responses, and to inject faults. This results in better reproducibility of tests and fault injections.

The possibility to conduct a test before the target hardware has been built is a further benefit of this approach. It makes testing possible very early in the development life cycle and supports the detection of development faults in the first step of the development cycle. This is a strong factor in dramatically reducing the cost induced by these faults [12].

On the other hand, the major drawback of this approach is the accuracy of the software model and, in particular, the compliance between this model and the actual hardware platform in terms of timing behavior. From a functional point of view, the purely software-based approaches can perfectly mimic the behavior achieved in the actual system. As far as the temporal aspects (e.g., the latencies, the achieved accuracy, etc.) are concerned, however, the purely software-based approaches reach their limit as soon as the desired accuracy of the tests and of the corresponding monitoring reaches the lower (finer grained) timing levels (see Section 12.3.1.1.2). Therefore, another important aspect is the cost of software-based testing approaches in terms of complexity of the model and computational resources for analyzing it. In any case, the accuracy of the software model has to be compliant with the granularity of the timing requirements of the application software, otherwise the results obtained by the performed test might not provide a sufficiently strong guarantee. While testing and monitoring on large scale timing levels (e.g., using a granularity of a single communication cycle) the computational resources required for the simulation are rather moderate when moving to lower (finer grained) timing levels; however, the computational resources may increase dramatically in terms of memory and CPU usage. Therefore, when applying purely software-based approaches, a trade-off has to be made between duration and the accuracy of a test run.

12.4.2 Hardware-Based Approaches

In hardware-based approaches, the cost factor related to the accuracy of a model, and therefore of its ability to fit to a given temporal granularity, is no longer a problem. In fact, there are no additional computational costs for a fine-grained level of timing requirement compared to a coarse-grained level.

However, hardware-based approaches are at a disadvantage in that they are less flexible and, in particular, the means of control over executing the test, monitoring, and fault injection is rather limited or comes at a great cost (e.g., expensive special purpose devices for reproducible deterministic fault injection [13]).

A further drawback with hardware-based approaches is the fact that usually the availability of the final target hardware is rather late in the development life cycle; this either requires the test to be conducted on early prototype samples (which exhibit flaws of their own) or to postpone the test until the final hardware is available. Both choices cause increased cost, either due to testing on immature hardware or due to late detection of development errors and thus increased cost for correcting and fixing these errors.

As mentioned previously, two methods can be used for hardware-based tests: open-loop tests and closed-loop tests. An example of functionality where a closed-loop based test is needed is complex communication services like TPs, NM, and diagnostic communication management. Each of these services implements a more or less complex communication protocol, requiring state machines at sender and receiver ends, where the state transitions are triggered by the messages received and/or transmitted. Without the possibility of reacting to the test responses of the DUT, the implementation of this kind of complex communication protocol in the test-bed is not possible. Another example is the testing of distributed control loops where the DUT is one of the ECUs participating in the control loop. Similar to the complex protocols, distributed control loops necessitate that the test-bed is capable of responding to the test responses of the DUT, which renders an open-loop approach inadequate in these situations.

In general, closed-loop approaches are more expensive (due to need for more computational resources and the fact that the tester itself has to be carefully designed and developed) than open-looped approaches. Therefore, in practice, a combination of both setups is used. Testing with closed-loop testing setups only takes place when previous tests conducted in cheaper (i.e., open loop) setups have successfully been completed.

12.5 Conclusion

Testing FlexRay-based systems has to take place at the interfaces of the different levels of abstraction of the hardware and software architecture, namely the PLI, the CHI, and the AI. Due to the nature of the FlexRay communication protocol, the responses of the DUT have to be monitored and examined, not only for deviations in the value domain, but also for deviations in the time domain and the code domain.

These tests have to be conducted in fault-free scenarios as well as under fault conditions, which must be caused by means of fault injection. The application of both pure software-based tests and hardware-based tests in open- and closed-loop test scenarios makes perfect sense, since each approach has its own merits and drawbacks. Each one has its own niche, where it provides the most benefits.

We therefore recommend using a combination of all these approaches with respect to the different development stages of the whole system. We propose starting with pure software-based approaches as long as no hardware is available. In the pure software-based approach, general functional tests can be conducted at rather large-scale timing levels. Once these tests have been completed successfully, additional tests at a smaller timing scale can be conducted.

Once hardware is available, open-loop tests can be conducted to verify the correct timing and functionality of the simple application parts of the DUT. Once these tests have been successfully completed as well, closed-loop testing approaches can be used to verify the complex protocols and the distributed control loop functionality.

References

1. T. Scharnhost, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Maté, and K. Nishikawa, AUTomotive Open System ARchitecture—an industry-wide initiative to manage the complexity of emerging automotive E/E-architectures, in *Convergence 2004, International Congress on Transportation Electronics*, Detroit, MI, 2004.
2. T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei, Timing analysis of the FlexRay communication protocol, in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE Computer Society, Washington, DC, July 5–7, 2006, pp. 203–216 (DOI = http://dx.doi.org/ 10.1109/ECRTS.2006.31).
3. B. Gaujal and N. Navet, Maximizing the robustness of TDMA networks with applications to TTP/C, *Real-Time Systems*, 31(1–3), 5–31, December 2005 (DOI = http://dx.doi.org/10.1007/s11241-005-2743-4).
4. ISO (International Organization for Standardization), *Road Vehicles—Diagnostics on Controller Area Networks (CAN)—Part 2: Network Layer Services*, ISO/DIS 15765-2.2, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, June 2003.
5. ISO (International Organization for Standardization), *Road Vehicles—Diagnostic Systems—Keyword Protocol 2000—Part 3: Application Layer*, ISO/DIS 14230-3, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 1999.

6. ISO (International Organization for Standardization), *Road Vehicles—Unified Diagnostic Services (UDS)—Part 1: Specification and Requirements*, ISO/DIS 14229-1, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, 2004.
7. R. Mores, G. Hay, R. Belschner, J. Berwanger, S. Fluhrer, E. Fuchs, B. Hedenitz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann, FlexRay—the communication system for advanced automotive control systems, in *Proceedings of SAE*, Paper 2001-01-0676.
8. FlexRay Consortium Web Page, http://www.flexray.com.
9. J. Berwanger and A. Schedl, BMW Group, and Ch. Temple, Freescale semiconductor, in *FlexRay Hits the Road*, Automotive DesignLine, November 15, 2006. Available at: http://www.automotivedesignline.com
10. J.-C. Laprie, B. Randell, A. Avizienis, and C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing*, 1(1), 11–33, 2004.
11. J.L. Welch and N.A. Lynch, A new fault-tolerant algorithm for clock synchronization, *Information and Computation*, 77(1), 1–36, April 1988.
12. G. Tassey, The economic impacts of inadequate infrastructure for software testing, NIST Report 02-3; National Institute of Standards and Technology, Acquisition and Assistance Division, Building 101, Room A1000, Gaithersburg, MD 20899-0001, USA; May 2002. Available at: http://www.nist.gov/director/prog-ofc/report02-3.pdf.
13. J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber, Comparison of physical and software-implemented fault injection techniques, *IEEE Transactions on Computers*, 52(9), 1115–1133, September 2003.

13

Timing Analysis of CAN-Based Automotive Communication Systems

Thomas Nolte
Mälardalen University

Hans A. Hansson
Mälardalen University

Mikael Nolin
Mälardalen University

Sasikumar Punnekkat
Mälardalen University

13.1 Introduction

The controller area network (CAN)* is one of the major fieldbus technologies, used in many application domains requiring embedded communications. CAN is particularly important in the automotive domain since it provides predictable temporal (real-time) behavior on message frame transmissions. This chapter goes into detail, presenting CAN, its history, and its properties. The chapter also describes timing analysis of CAN frames.

13.1.1 History

In the beginning of the 1980s, Robert Bosch GmbH evaluated existing serial bus systems (network technologies) in terms of usability in the automotive domain. None of the existing technologies were suitable. As a consequence, in 1983 Bosch proposed the CAN. This new network technology was primarily intended to support adding of new functionality in automotive systems. Moreover, replacing dedicated cables with a shared network also reduces the cabling, an issue of growing importance in vehicles. In February 1986, Bosch presented "Automotive Serial Controller Area Network" at the SAE congress in Detroit, and CAN was officially born. The following year, the first CAN communication adapter chips were released by Intel and Philips. However, it was not until the beginning of the 1990s that Bosch submitted the CAN specification for international standardization. At the end of 1993, CAN was standardized by ISO as ISO standard 11898 [1]. At the same time, a low-speed fault-tolerant physical layer version of CAN was standardized as ISO 11519-2 [2]. Two years later, ISO 11898 was extended with an addendum to also include an extended version of CAN.

Looking at CAN today, there is a wide variety of CAN standards adapted to the demands of different domains: ISO 11898 is the most commonly used fieldbus in the European automotive industry. In the United States, however, different CAN-based standards such as the SAE J1850 [3] are more common, although it is now being replaced [4] by SAE J2284 [5]. Also, for trucks and trailers, the SAE J1939 [6] is used since the late 1990s. The SAE J1939 was published by SAE in 1998, as a result of the work initiated in the early 1990s by the SAE truck and bus control and communications subcommittee. J1939 specifies how message frames are defined for engine, transmission, and brake systems in truck and trailer applications. Nowadays, SAE J1939 is widely used in truck and trailer applications, and standardized as ISO 11992. Finally, looking at other application domains, for tractors and machinery for agriculture and forestry, an SAE J1939-based ISO standard is used: ISO 11783 [7,8] and NMEA 2000 [9] define an SAE J1939/ISO 11783-based protocol for marine usage.

In the remainder of this chapter, all above-mentioned CAN standards are referred to as CAN for simplicity, as their differences mainly are in speed and usage of frame identifiers.

* Robert Bosch GmbH, BOSCH's controller area network, http://www.can.bosch.com/.

13.1.2 Applications

A typical CAN application is any type of distributed embedded system with real-time requirements and cycle times of 5 ms or more. However, CAN is used for many non-real-time applications as well. CAN was first used in the automotive industry by BMW, in their 850 CSI model of 1986. Mercedes-Benz introduced fieldbuses in their SL 500 model of 1989, and in 1992 CAN was also used in their bigger S series. Typically, initially one CAN bus was used for engine control, but as a second step, a gateway was introduced connecting the engine control network with another CAN controlling body and comfort electronics. For a good overview of applications where CAN is used interested readers are referred to the CAN in Automation (CiA) Web site.*

13.1.3 Chapter Organization

Section 13.2 presents CAN, including basic properties such as network topologies, frame types, arbitration mechanism, error detection and bit-stuffing mechanisms, and frame transmission time. The standard CAN message frame scheduler is presented along with other CAN schedulers in Section 13.3. Section 13.4 presents the scheduling model used in this chapter, followed by methods for calculating worst-case message frame response times in Section 13.5. Error models and modified response-time analysis are introduced in Section 13.6. Section 13.7 presents holistic analysis, allowing for analysing end-to-end temporal behavior of CAN-based systems. Section 13.8 presents middlewares and frame packing, commonly used in the automotive domain. Finally, the chapter is summarized in Section 13.9.

13.2 CAN

CAN is a broadcast bus, which uses deterministic collision resolution (CR) to control access to the bus (so-called carrier sense multiple access [CSMA]/CR).

In general, CSMA protocols check the status of the medium before transmitting a frame [10], to see if the medium is idle or busy (this process is called carrier sensing). By only initiating frame transmissions when the medium is idle, CSMA protocols allow for ongoing frame transmissions to be completed without disturbance. If the medium is busy, CSMA protocols wait for some time before a transmission is tried again. Specifically, CAN waits until the medium becomes idle, and the frame arbitration mechanism (detailed below) resolves potential collisions when multiple frames are arbitrated in parallel.

CAN transmits "messages" using "frames" containing 0–8 bytes of "payload data" and a number of "control bits." These frames can be transmitted at speeds of 10 kbps up to 1 Mbps. Depending on the CAN format, standard or extended, the number of control bits is either 47 or 67. Specifically, the number of identifier bits is either 11 (CAN standard format) or 29 bits (CAN extended format). The various parts of the standard format CAN frame are shown in Figure 13.1.

* CAN in Automation (CiA), http://www.can-cia.org/.

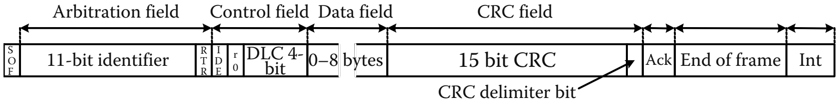


FIGURE 13.1 CAN frame layout (CAN standard format data frame).

13.2.1 Topology

The CAN protocol operates on a broadcast bus. There is no built-in support for other topologies. However, third-party solutions with gateways or switches for star topologies exist [11–13], see also Chapter 6.

In advanced applications, such as an automotive system, several CAN busses are interconnected using one or more gateways. A gateway is provided with software that knows which data should be forwarded between the different busses it is connected to. Using gateways, arbitrary large and complex CAN-based networks can be constructed. However, the gateways have to be manually programmed to provide the gateway functions.

As one example, a typical automotive CAN-based network architecture is depicted in Figure 13.2. The figure shows the gatewayed network infrastructure of a VW Passat, reproduced and based upon material presented in Ref. [14]. Note here that several CAN nodes are acting as gateways to low-cost (local interconnect network) LIN [15, 16] as well.

From a real-time perspective it should be noted that each gateway will add delays to the transmission of the frame, potentially causing the frame to miss its deadline. Also, when using a gateway all nodes may not receive a specific frame at the same time (since the frame can experience different delays on different segments), making it more difficult to implement tightly coordinated behavior in a distributed system.

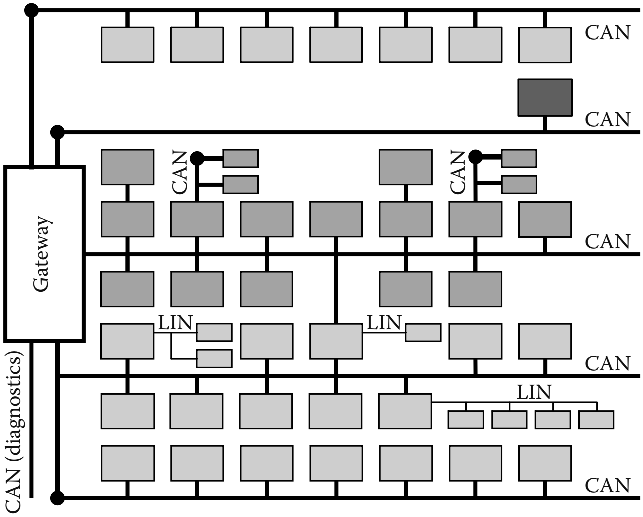


FIGURE 13.2 Network infrastructure of VW Passat.

More information and further pointers on issues related to CAN gateways are found in Ref. [13].

13.2.2 Frames

The CAN protocol operates using four kinds of frames:

1. Data frames
2. Remote transmit request (RTR) frames
3. Overload frames
4. Error frames

In this chapter, focus is given to data frames, for which real-time analysis techniques are presented in Section 13.5. This analysis is extended in Section 13.6 to also include the effects of errors. However, overload frames are not included in this chapter. Although outside the scope and purpose of this chapter, the presented response-time analysis can easily be extended to include RTR frames using the approach outlined in Ref. [17].

13.2.3 Frame Arbitration

CAN implements CR by frame arbitration, the process of selecting the frame with the highest priority (which is equivalent to the frame with the lowest identifier) among a set of frames that are simultaneously sent by a set of nodes. Besides selecting the highest priority frame, the arbitration guarantees that the CAN bus is collision free. In order to give this guarantee, the CAN protocol requires that two simultaneously active data frames originating from different source nodes must have different identifiers. In summary, the identifier serves the following purposes:

1. Identifying the frame
2. Assigning a priority to the frame
3. Enabling receivers to filter frames

The physical layer of CAN makes sure that (1) an idle CAN bus has the logical value of 1, and (2) if any node is sending a 0 it will result in a bus value 0. Thus, if two (or more) nodes simultaneously try to send bits, the bus will only have the value 1 if all nodes send 1. Figure 13.3 illustrates this principle when two nodes simultaneously send bit streams on the bus.

Using this property, the frame arbitration is implemented by each node simultaneously listening to the bus while sending its identifier. If the bus value is different from the identifier bit that is just being written, then the node knows that there is a higher priority frame trying to access the bus and thus it stops transmitting.

Figure 13.4 shows the scenario when four nodes simultaneously try to send a frame. Initially, the bus is idle. Each node knows this since the bus value is 1. Next, each node starts transmission with the start of frame (SOF) bit (see Figure 13.1). Since each node writes 0 and also reads 0, no node can know that other nodes try to access the bus at the same time. Now, each node starts transmitting the bits of their identifiers. For

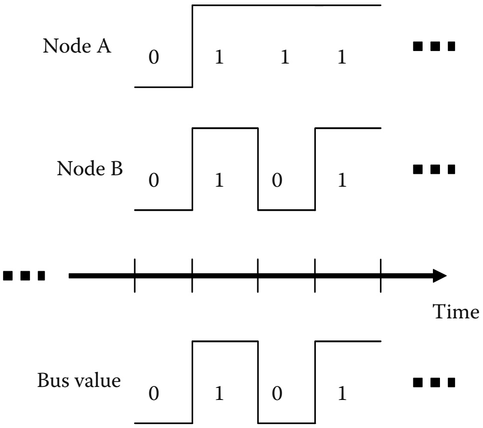


FIGURE 13.3 Bus value with multiple communication adapter transmissions.

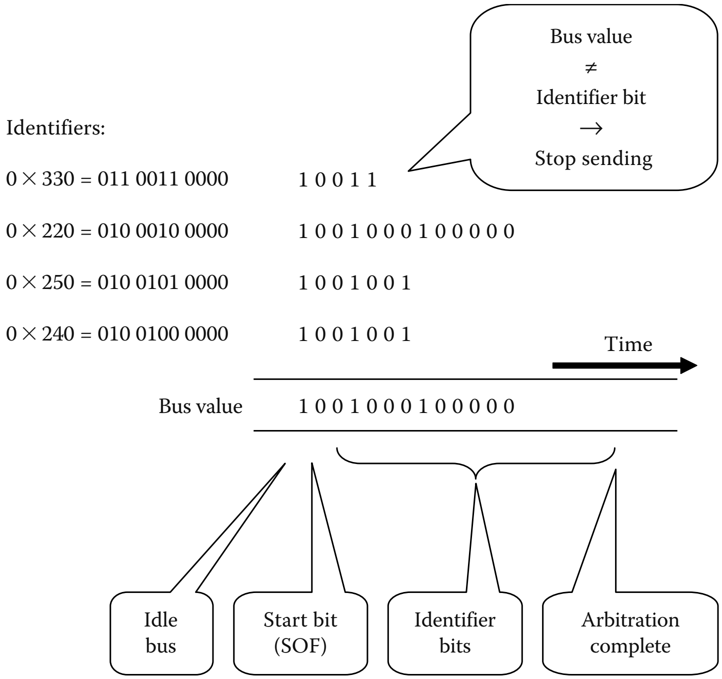


FIGURE 13.4 Arbitration among four simultaneous frames.

the two first bits, again, each node reads the same value as it transmits and thus they all continue to send identifier bits. However, at the third identifier bit the node with frame-id 0 × 330 writes a 1 but reads a 0. At this time that node knows it is not sending the highest priority frame and thus it stops sending (still monitoring the bus and waiting for it to become idle after the transmission of the current frame). For the other three nodes the arbitration continues until the fifth identifier bit, when the two nodes with frame identifiers 0 × 240 and 0 × 250 realize that none of them have the highest priority frame. The node with frame identifier 0 × 220, however, can continue to send its complete frame identifier. When the complete identifier has been transmitted one single node knows that it is sending the highest priority frame, and that all other nodes have stopped transmitting. Thus, it can now continue sending the rest of frame, knowing that there will be no collision on the bus.

Relying on the CAN arbitration mechanism, CAN implements CSMA/CR, behaving like a global priority-based queue. It is worth noting that, any time, regardless of the number of nodes that enter arbitration, the arbitration time is constant. Thus, CAN is a highly effective implementation of a distributed priority queue. Hence, CAN behaves like a fixed priority non-preemptive system, that is, once a node has won arbitration it will always fully transmit its frame, and any higher priority frames that may arrive during transmission must wait until the next arbitration round starts after transmission completes.

13.2.4 Error Detection

In CAN, errors may occur due to electromagnetic interference (EMI) from the operational environment, different sampling points or switching thresholds in different nodes, or due to signal dispersion during propagation. To handle these scenarios, the CAN protocol provides elaborate error detection and self-checking mechanisms [18], specified in the data link layer of ISO 11898 [1]. The error detection is achieved by means of transmitter-based monitoring, bit-stuffing, cyclic redundancy check (CRC), and frame check.

To make sure that all nodes have a consistent view, errors detected in one node must be globalized. This is achieved by letting the detecting node transmit an error flag containing 6 bits of same polarity. Upon reception of an error frame, each node will discard the erroneous frame, which then will be automatically retransmitted by the sender. Note that, the retransmitted frame could be subjected to arbitration during retransmission. This implies that if any higher priority frame gets queued during the transmission and error signaling of the current frame, then those frames will be transmitted before the erroneous frame is retransmitted.

Specification documents of CAN [19] claim that the error detection mechanisms can detect and globalize all transmitter errors. Bursts are guaranteed to be detected on the receiver side up to a length of 15 (which is equal to the degree of $f(x)$ in CRC sequence). Most longer error bursts are also detected. The probability for undetected errors is negligibly small, as per the CAN technical information [19], which states that "with an operating time of eight hours per day on 365 days per year and an error rate of 0.7 s, one undetected error occurs every thousand years (statistical average)." Of late, there have been studies indicative of the limitations of CAN for applications with ultrahigh dependability requirements, especially on its vulnerabilities due to undetected multibit errors [20] as well as on the validity of underlying assumptions of CRCs [21] (see also Chapter 6). Even though there is a positive probability for

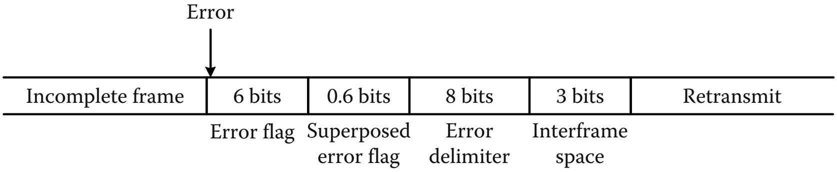


FIGURE 13.5 Error frame format in CAN.

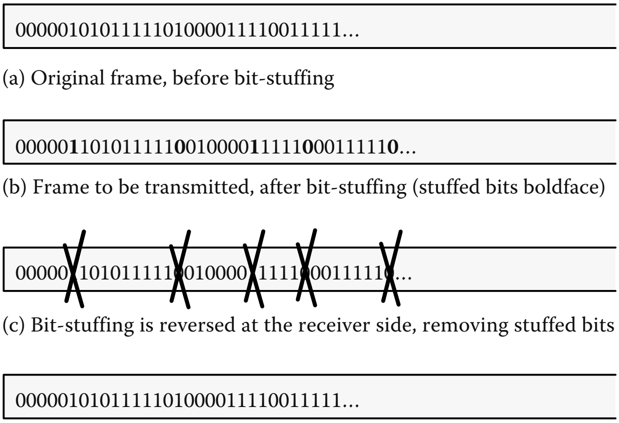
undetected errors, for simplifying the presentation, we shall assume that all errors are detected.

Figure 13.5 shows formats of the CAN error frames (details are given in Ref. [19]). It can be seen that error signaling and recovery time are typically between 17 and 23 bit times. Since we are interested in the worst-case behavior, we shall use 23 bit times as the error signaling and recovery time in our model.

13.2.5 Bit-Stuffing

As described above, six consecutive bits of the same polarity (111111 or 000000) are used for error signaling. To avoid these special bit patterns in the contents of transmitted frames, a bit of opposite polarity is inserted after each occurrence of five consecutive bits of the same polarity. By reversing the procedure, these bits are removed at the receiver side. This technique depicted in Figure 13.6a through d, which is called "bit-stuffing," implies that the actual number of transmitted bits may be larger than the size of the original frame, corresponding to an additional transmission delay that needs to be considered in the analysis.

Looking at a CAN frame, the number of bits, beside the data part in the frame, that are exposed to the bit-stuffing mechanism is defined as $v \in \{34, 54\}$, depending on the



(a) Original frame, before bit-stuffing

(b) Frame to be transmitted, after bit-stuffing (stuffed bits boldface)

(c) Bit-stuffing is reversed at the receiver side, removing stuffed bits

(d) Transmitted frame, same as original

FIGURE 13.6 Bit-stuffing example; from sender to receiver.

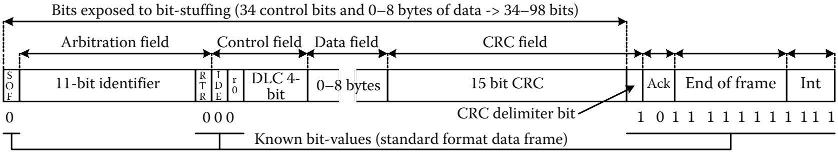


FIGURE 13.7 Bits subject to bit-stuffing (CAN standard format data frame).

1111100001111000011110000111100001111...

(a) Original frame, before bit-stuffing

11111**0**0000**1**1111**0**0000**1**1111**0**0000**1**1111**0**0000**1**1111...

(b) Frame to be transmitted, after bit-stuffing (stuffedbits **boldface**)

FIGURE 13.8 The worst-case scenario when stuffing bits.

CAN format used: either 34 (CAN standard format) or 54 (CAN extended format) bits. Note that 13 bits in the CAN frame (including the 3 bit interframe space) are not exposed to the bit-stuffing mechanism (Figure 13.7).

The worst-case number of stuffed bits in an arbitrary stream of bits is given by

$$\mathtt{ns}(n) = \left\lfloor \frac{n-1}{4} \right\rfloor \tag{13.1}$$

where n is the number of bits in the stream of bits. $\lfloor a/b \rfloor$ is notation for the floor function, which returns the largest integer less than or equal to a/b. Intuitively, Equation 13.1 captures the worst-case bit-stuffing of a stream of bits depicted in Figure 13.8. Note that, in the example, as soon as a bit is stuffed, it gives rise to a new five consecutive bits (a sequence of bits not present in the original stream of bits), hence producing yet another stuffed bit, after which the scenario is repeated again, etc.

13.2.6 Frame Transmission Time

The number of bytes of "payload data" in a CAN frame i is defined as $s_i \in [0, 8]$. Recall that a CAN frame can contain between 0 and 8 bytes of payload data. Hence, looking at Section 13.2.5 and Figure 13.7, let c_i give the total number of bits in a CAN frame before bit-stuffing as

$$c_i = v + 13 + 8s_i \tag{13.2}$$

where v and 13 are the number of bits in the CAN frame control bits exposed and not exposed, respectively, to the bit-stuffing mechanism, and $8s_i$ is the number of payload data bits.

Let τ_{bit} be the time taken to transmit one bit on the bus—the so-called "bit-time." Now, considering that only $v + 8s_i$ bits in the CAN frame are subject to bit-stuffing, the worst-case time C_i taken to transmit a given frame i is given by

$$C_i = \left(v + 13 + 8s_i + \left\lfloor \frac{v + 8s_i - 1}{4} \right\rfloor \right) \tau_{\text{bit}} \qquad (13.3)$$

For CAN standard format frames (11-bit identifiers), Equation 13.3 simplifies [22] to

$$C_i = (55 + 10s_i)\tau_{\text{bit}} \qquad (13.4)$$

and for CAN extended format frames (29-bit identifiers), Equation 13.3 can be simplified to

$$C_i = (80 + 10s_i)\tau_{\text{bit}} \qquad (13.5)$$

13.3 CAN Schedulers

On top of the CAN frame arbitration mechanism, a number of frame schedulers have been proposed for CAN. In general, these schedulers can be divided into three groups: time-driven, priority-driven, and share-driven schedulers.

CAN provides physical signaling, medium access control (MAC), and addressing (via identifiers). These are the two lowest layers of the open systems interconnection (OSI)-layered model for communication protocols [23,24]. Since the services provided by CAN are rather basic, and only provide one type of frame scheduling (i.e., fixed priority scheduling [FPS], as discussed in Section 13.2.3), several protocols have been developed on top of CAN, both in the academia as well as in the commercial domain.

By introducing a higher layer protocol running on top of CAN, it is possible to achieve a number of different schedulers. Original (native) CAN is suitable for handling periodic real-time traffic according to the FPS approach. Limiting CAN to a periodic traffic model, timing analysis can easily be applied and schedulability checked. However, due to the limitations inherent in FPS scheduling, adaptations to allow other scheduling policies have been developed. As an alternative to the fixed-priority mechanisms offered by native CAN, some higher layer protocols have been developed to implement priority-driven dynamic priority scheduling (DPS) schedulers (such as earliest deadline first [EDF]), time-driven schedulers, and share-driven schedulers.

Several options for time-driven scheduling of CAN exist. These protocols typically implement a master/slave mechanism, having a central master node controlling the network in a time-driven fashion. An example of a time-driven scheduler for CAN is TT-CAN [25]. Also, FTT-CAN [26,27] provides time-driven scheduling as well as the option to combine time-driven and priority-driven scheduling. More information on these schedulers can be found in Chapter 6.

By manipulating the CAN frame identifier online, and therefore changing the frame priority dynamically, several approaches to mimic EDF type of scheduling have been presented [28–31]. However, by manipulating the identifier of the CAN frames, all these solutions reduce the number of possible identifiers to be used by the system designers. This could be problematic, since it interferes with other design activities, and is even sometimes in conflict with adopted standards and recommendations [32,33].

A common way to send non-real-time frames on CAN is to allocate frame identifiers with lower priority than all real-time frames. In this way, it can be made sure that a non-real-time frame can block a real-time frame at most for the duration of the transmission of one frame. However, unwise frame identifier assignment to non-real-time frames could cause some of them to suffer from starvation. To provide quality of service (QoS) for non-real-time frames several approaches have been presented [34,35]. These approaches dynamically change frame identifiers in a way preventing systematic penalization of some specific frames.

By providing the option of share-driven scheduling of CAN, designers are given more freedom in designing a distributed real-time system where the nodes are interconnected with a CAN. As time-driven schedulers, share-driven schedulers are implemented using a master/slave mechanism running on top of CAN, scheduling the network according to share-driven scheduling policies. Server-CAN [36–38] is a share-driven scheduler for CAN. See Chapter 6 for more details on Server-CAN.

Looking at the context of this chapter, focus is given to response-time calculations of automotive CAN-based systems. Hence, most schedulers outlined above should merely be considered as related work in the domain of CAN schedulers, rather than the type of schedulers used in automotive systems. In the remainder of the chapter, attention is given to priority-driven scheduling, which is the most natural scheduling method since FPS is the policy implemented by the CAN arbitration mechanism. Analyses have been presented to determine the schedulability of CAN frames [22].* This analysis is based on the standard FPS response-time analysis for CPU scheduling [39].

13.4 Scheduling Model

Most CAN-based systems have hard real-time requirements and CAN is used as the communication network due to its inherent properties of providing real-time guarantees of frame transmissions. In order to reason about the temporal behavior of individual frame transmissions, a predictable model is used. In Section 13.5, it is shown how to use frame response-times in order to determine the schedulability of a CAN-based system. The model used in this chapter is based on frame priorities, frame periods, frame deadlines, and frame response times.

Using the CAN frame identifier, the CAN frame arbitration mechanism provides collision-free transmission of frames. Here, the priority of a frame is determined by the frame identifier, and a numerically lower value frame identifier reflects a higher frame priority. The identifier of a CAN frame is fixed, so is the frame priority. Hence, i denotes both priority and identifier of a frame.

* Originally presented in Refs. [17,40,41] but revised in Ref. [22].

Frame transmission on the CAN bus is non-preemptive. When the CAN bus is transmitting a frame the bus is busy, transmitting the frame until completion, and when it is not transmitting a frame the bus is idle. As soon as the bus is idle, the transmission of a CAN frame may initiate. The CAN arbitration mechanism enforces transmission of the highest priority available frame as soon as the bus becomes idle.

For the analysis presented in this chapter, frames are assumed to be generated in a periodic manner, that is, with a fixed amount of time between any instances of the same frame. Hence, each frame is associated with a frame period denoted T_i, that is, the time between the generation of two instances of the same frame. This temporal representation also models sporadic frame arrivals, that is, frames that are not generated periodically but have a known minimum interarrival time between instances of the same frame. The sporadic frames are simply represented by their worst-case arrival pattern, corresponding to periodic frames with periods (T_i) equal to the minimum interarrival time of the corresponding sporadic frame.

As frames are generated by software running on the nodes in the system, an additional time needs to be taken into account to cater for queuing and interrupt latencies. In fact, the time between the creation of the frame in the software at the node, and the time when the frame is available in the CAN communication adapter, is called the "queuing jitter." This time is assumed to be bounded by J_i.

Each frame i has an associated frame relative deadline, denoted D_i. A frame relative deadline is the time which it must have completed transmission before, relative to the start of the frame period T_i.

Finally, the frame worst-case response time R_i, is the longest time needed for any instance of frame i to finish frame transmission, relative to the start of the period T_i. The system is said to be schedulable if $R_i \leq D_i \forall i$.

13.5 Response Time Analysis

Calculating the worst-case response-times requires a bounded worst-case queuing pattern of frames. The standard way of expressing this is to assume a set of traffic streams, each generating frames with a fixed priority. The worst-case behavior of each stream, in terms of network load, is to send as many frames as they are allowed, that is, to periodically queue frames in their corresponding communication adapter. Similar to CPU scheduling, a model with a set $\mathcal{S}$ of streams (corresponding to frame transmitting CPU tasks) is used. Each $S_i \in \mathcal{S}$ is a tuple $<P_i, T_i, J_i, C_i>$, where P_i is the priority (defined by the frame identifier), T_i is the period, J_i is the queuing jitter, and C_i is the worst-case transmission time of frame i.

In this section, two approaches for the calculation of the worst-case response-time R_i of a CAN frame i sent on stream S_i are presented. The first is a simple, "sufficient" response-time test. The second one, which is slightly more complex than the first one, is a "sufficient and necessary" worst-case response-time test.

A response-time calculation test is sufficient when all frames that are calculated to be schedulable are in fact schedulable. If a response-time calculation test is sufficient, but "not necessary," this means that a frame deemed not schedulable might in fact be schedulable. Hence, using a sufficient and necessary worst-case response-time test, more frames are likely to be classified as schedulable, although at a cost of using a more complex worst-case response-time calculation. Below, we denote the sufficient and necessary worst-case response-time test for CAN as an "exact response-time test."

The concept of a busy period [42] in the context of frame communication is defined in Ref. [22] as an interval starting at some time t^{s} when a frame with priority i or higher is queued for transmission, and there are no frames of priority i or higher waiting to be transmitted that were queued strictly before time t^{s}. The interval is contiguous, preventing transmission of any frame with priority lower than i, ending at time t^{e}, when there are no frames of priority i or higher waiting to be transmitted that were queued strictly before time t^{e}.

A key characteristic of the busy period is that any frame, queued strictly before the end of the busy period t^{e}, with priority higher than or equal to i, is sent during the level-i busy period. The end of a busy period may correspond to the start of another busy period.

13.5.1 Sufficient Response-Time Test

The worst-case response-time of a frame is found in the busy period beginning with a critical instant [43], where all frames of priority i and higher are simultaneously queued at their corresponding communication adapters. Following this, as outlined in Ref. [22], the worst-case response-time for frame i is given by

$$R_i = J_i + w_i + C_i \tag{13.6}$$

where J_i is the queuing jitter of frame i, that is, the maximum variation in queuing-time relative to the start of the frame period T_i, inherited from the sender task that queues the frame, C_i is the worst-case transmission time of frame i (given by Equation 13.3), and w_i is the queuing delay given by solving the equation

$$w_i^{n+1} = B^{\mathrm{MAX}} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n + J_j + \tau_{\mathrm{bit}}}{T_j} \right\rceil C_j \tag{13.7}$$

where

B^{MAX} corresponds to the transmission time of the longest possible CAN frame (i.e., the worst-case transmission time of a CAN frame with 8 bytes of payload data)

$hp(i)$ is the set of frames with priority higher than that of frame i

Note that Equation 13.7 is a recurrence relation with an initial value that can be chosen equal to C_i, that is, $w_i^0 = C_i$, and a terminating condition of either $w_i^{n+1} = w_i^n$ and/or $J_i + w_i^{n+1} + C_i > D_i$. Only if the latter condition is false, the frame is schedulable.

13.5.2 Exact Response-Time Test

For many applications, it is sufficient using the slightly pessimistic response-time calculation given by R_i in Equation 13.6. However, in search of the exact worst-case response-time $\overline{R}_i$ of a CAN frame i sent on stream S_i, it is found within what is called the level-i busy period. Specifically, the individual frame response-time has to be calculated for each instance of frame i within its busy period [22]. The worst-case response-time $\overline{R}_i$ is experienced by one or more of the instances of a specific frame within its corresponding busy period. Hence, in order to derive the exact worst-case response-time, the response-time has to be calculated for all these frame instances.

As a first step, the length t_i of the level-i busy period is given by solving the following recurrence relation, starting with an initial value of $t_i^0 = C_i$, and finishing when $t_i^{n+1} = t_i^n$

$$t_i^{n+1} = B_i + \sum_{\forall k \in \text{hep}(i)} \left\lceil \frac{t_i^n + J_k}{T_k} \right\rceil C_k \tag{13.8}$$

where

B_i is the maximum blocking time due to lower priority frames in the process of frame transmission
hep(i) is the set of frames with priority higher than or equal to that of frame i
J_k is the queuing jitter of frame k, that is, the maximum variation in queuing-time relative to the start of the frame period T_k, inherited from the sender task that queues the frame
C_k is the frame transmission time for frame k derived from Equation 13.3

Now, looking at a specific frame i, the number of instances Q_i of frame i that become ready for frame transmission before the end of the busy period is given by

$$Q_i = \left\lceil \frac{t_i + J_i}{T_i} \right\rceil \tag{13.9}$$

For each instance $(0, \dots, Q_i - 1)$ of frame i, the corresponding worst-case frame response-time must be derived. Letting q being the index of a frame instance, the worst-case response-time $\overline{R}_i(q)$ of frame instance number q is given by

$$\overline{R}_i(q) = J_i + \overline{w}_i(q) - qT_i + C_i \tag{13.10}$$

where $\overline{w}_i(q)$ represents the effective queuing time, given by the recurrence relation in Equation 13.11, starting with an initial value of $\overline{w}_i^0(q) = 0$, and finishing when $\overline{w}_i^{n+1}(q) = \overline{w}_i^n(q)$ or when $J_i + \overline{w}_i^{n+1}(q) - qT_i + C_i > D_i$ (i.e., either when a worst-case response-time is found, or when the frame is found not schedulable). In Equation 13.11, τ_{bit} is the bit-time.

$$\overline{w}_i^{n+1}(q) = B_i + qC_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{\overline{w}_i^n(q) + J_j + \tau_{\text{bit}}}{T_j} \right\rceil C_j \tag{13.11}$$

Once all Q_i worst-case response-times are calculated, the worst-case response time $\overline{R}_i$ for frame i is found as the maximum response-time among these Q_i instances' response-times as follows:

$$\overline{R}_i = \max_{q=0,\ldots,Q_i-1} \left[\overline{R}_i(q)\right] \tag{13.12}$$

Note that this exact analysis is also valid for frames with deadlines greater than their period, which is not the case for the sufficient response-time test outlined in Section 13.5.1.

13.5.3 Example

Here, an example of the above analyses is given in order to illustrate their differences. More precisely, a set $\mathcal{S}$ containing three frames will be deemed not schedulable using the sufficient response-time test presented in Section 13.5.1, and schedulable using the exact response-time test presented in Section 13.5.2.

Consider the frame set outlined in Table 13.1. The basic assumptions on the system is that the bus speed is 1 Mbps, that is, $\tau_{\text{bit}} = 1\,\mu\text{s}$, and there is no jitter, that is, $J = 0$ for all frames. All frames have 3 bytes of payload data, hence, according to Equation 13.3, $C = 75$. The bus utilization is rather high, 97%. The reason for having such a high bus utilization is to force a delicate scenario not suitable for the sufficient analysis, but captured by the exact analysis.

13.5.3.1 Sufficient Response-Time Test

Starting with the sufficient response-time test, using Equation 13.7 to calculate w_i together with Equation 13.6 to calculate R_i for each frame i gives the following (recall that Equation 13.7 is a recurrence equation terminating when $w_i^{n+1} = w_i^n$):

$$w_1^0 = 0, \quad w_1^1 = 75, \quad w_1^2 = 75$$
$$R_1 = 150$$
$$w_2^0 = 0, \quad w_2^1 = 150, \quad w_2^2 = 150$$
$$R_2 = 225$$
$$w_3^0 = 0, \quad w_3^1 = 225, \quad w_3^2 = 300, \quad w_3^3 = 375, \quad w_3^4 = 450, \quad w_3^5 = 450$$
$$R_3 = 525$$

The above results say that frames 1 and 2 are schedulable, that is, $R_1 \le D_1$ and $R_2 \le D_2$. However, $R_3 > D_3$ indicates that a deadline is missed. Hence, the frame set $\mathcal{S}$ is deemed not schedulable.

TABLE 13.1 Frame Set under Analysis

Frame i	P_i	T_i	D_i	C_i
1	1	187.5	187.5	75
2	2	262.5	262.5	75
3	3	262.5	262.5	75

13.5.3.2 Exact Response-Time Test

Instead, by using the exact analysis of Section 13.5.2 the above frame set $\mathcal{S}$ is in fact schedulable. To show this, Equations 13.8 through 13.12 are used to calculate the frame response-times for all three frames.

As a fist step, using Equation 13.8, the level-i busy period has to be calculated for each frame to see, using Equation 13.9, if multiple instances of that frame are present within the busy period. If that is the case, the response-time has to be calculated for all these frame instances using Equations 13.10 and 13.11, and the maximum response-time among all instances is to be selected using Equation 13.12.

Starting with frame 1, using Equation 13.8, the level-1 busy period is calculated to be $t_1^0 = 75, t_1^1 = 150, t_1^2 = 150$. Now, $Q_1 = 1$ according to Equation 13.9. Hence, there is only one instance of frame 1 within the level-1 busy period. For this instance, the response-time is calculated using Equations 13.10 and 13.11:

$$\begin{aligned}&\overline{w}_1^0(0) = 0, \quad \overline{w}_1^1(0) = 75, \quad \overline{w}_1^2(0) = 75\\&\overline{R}_1(0) = 150\end{aligned}$$

For frame 2, the same procedure is repeated using Equation 13.8 giving $t_2^0 = 75$, $t_2^1 = 225, t_2^2 = 300, t_2^3 = 375, t_2^4 = 450, t_2^5 = \mathbf{450}$; hence $Q_2 = 2$ according to Equation 13.9, that is, there are two instances of frame 2 in the level-2 busy period. Here, the response-time is calculated for both these instances using Equations 13.10 and 13.11, and the maximum is selected using Equation 13.12:

$$\begin{aligned}&\overline{w}_2^0(0) = 0, \quad \overline{w}_2^1(0) = 150, \quad \overline{w}_2^2(0) = 150\\&\overline{R}_2(0) = 225\\&\overline{w}_2^0(1) = 0, \quad \overline{w}_2^1(1) = 225, \quad \overline{w}_2^2(1) = 300, \quad \overline{w}_2^3(1) = 300\\&\overline{R}_2(1) = 112.5\\&\quad \overline{R}_2 = 225\end{aligned}$$

Finally, for frame 3 there are two instances within the level-3 busy period as $t_3^0 = 75$, $t_3^1 = 225, t_3^2 = 300, t_3^3 = 450, t_3^4 = 525, t_3^5 = 525$ and $Q_3 = 2$. Following the same reasoning as for frame 2, the response-time is calculated:

$$\begin{aligned}&\overline{w}_3^0(0) = 0, \quad \overline{w}_3^1(0) = 150, \quad \overline{w}_3^2(0) = 150\\&\overline{R}_3(0) = 225\\&\overline{w}_3^0(1) = 0, \quad \overline{w}_3^1(1) = 225, \quad \overline{w}_3^2(1) = 300, \quad \overline{w}_3^3(1) = 375,\\&\overline{w}_3^4(1) = 450, \quad \overline{w}_3^5(1) = 450\\&\overline{R}_3(1) = 262.5\\&\quad \overline{R}_3 = 262.5\end{aligned}$$

To summarize, using the exact analysis rather than the sufficient one, the whole frame set is schedulable as $\overline{R}_1 \leq D_1$, $\overline{R}_2 \leq D_2$, and $\overline{R}_3 \leq D_3$. The sufficient test is much faster as it requires fewer calculations to be performed. In most cases, the sufficient test yields the same frame sets schedulable, compared with the exact analysis. However, there are cases, as shown in the example above, when only the exact analysis will show that a set of frames is in fact schedulable.

13.6 Timing Analysis Incorporating Error Impacts

The model underlying the basic CAN response-time analysis presented above assumes an error-free communication bus, that is, all frames sent are assumed to be correctly received, which in reality is not always true. For instance, in applications such as automobiles, the systems are often subjected to high degrees of EMI from the operational environment which may cause transmission errors on the CAN bus. The common causes for such interference include cellular phones and other radio equipments inside the vehicle and electrical devices like switches and relays as well as radars, radio transmissions from external sources, and lightning. It has not been possible to completely eliminate the effects of EMI since exact characterization of all such interferences defies comprehension. Though usage of an all-optical network could greatly eliminate EMI problems, it is not favored by the cost-conscious automotive industry.

These interferences cause errors in the transmitted data, which could indirectly lead to catastrophic results. To reduce the risk for such results, CAN designers have provided elaborate error checking and confinement features in the protocol. Basic philosophy of these features is to identify an error as fast as possible and then retransmit the affected frame. However, the effect will be increased frame latencies, which may lead to missed deadlines, especially if the interference coincides with the worst-case frame transmission scenario considered when performing schedulability analysis. This can be problematic in systems without spatial redundancy of communication medium/controllers, and the fault-tolerance mechanism employed is only time redundancy, where increased latencies of frame sets potentially could lead to violation of timing requirements.

Thus, the worst-case response-time calculations presented in Section 13.5 must be extended to handle the effect of various errors occurring in the channel. Over the years, a number of error models along with modified response-time analysis have been developed. The first error model was presented by Tindell et al. [17], however, only modeling strictly periodic interference.

13.6.1 Simple Error Model

Assuming a simple error model, relying on a function $F(t)$ representing the maximum number of errors that can be present during a time interval t, the response-time calculations above can be extended to handle faults. Recall that the worst-case implication of an error is the transmission of a 23-bit error frame (see Section 13.2.4). Following the error frame, the frame subjected to the error will also be retransmitted by the CAN communication adapter. Hence, looking at a specific frame i, the maximum delay caused by an error, during a time interval of t, is given by

$$E_i(t) = \left[23\tau_{\text{bit}} + \max_{k \in \text{hep}(i)}(C_k)\right] F(t) \tag{13.13}$$

where hep(i) is the set of frames with priority higher than or equal to that of frame i.

13.6.2 Modified Response-Time Analysis

Assuming the error model given by Equation 13.13, a simple adjustment can be made to Equations 13.7 and 13.11 in order to also cover for faults, as follows

$$w_i^{n+1} = E_i(w_i^n + C_i) + B^{\text{MAX}} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i^n + J_j + \tau_{\text{bit}}}{T_j} \right\rceil C_j \tag{13.14}$$

and

$$\overline{w}_i^{n+1}(q) = E_i(\overline{w}_i^n + C_i) + B_i + qC_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{\overline{w}_i^n(q) + J_j + \tau_{\text{bit}}}{T_j} \right\rceil C_j \tag{13.15}$$

Again, looking at these recurrence equations, initial values are set to $w_i^0 = C_i$ and $\overline{w}_i^0(q) = C_i$, respectively. They have the same termination condition as Equations 13.7 and 13.11, and they are guaranteed to converge provided that the network utilization including error recovery overhead is less than 100% [22].

13.6.3 Generalized Deterministic Error Model

The error model presented above is very simple and thus not really appropriate to describe real faults. When verifying a system with respect to EMI, multiple sources of errors have to be considered. Handling of each source separately is not sufficient; instead they have to be composed into a worst-case interference with respect to latency on the bus. Also, each source can typically be characterized by a signaling pattern of shorter or longer bursts, during which the bus is unavailable, that is, no signaling will be possible on the bus.

A generalized deterministic model was presented by Punnekkat et al. [44], which has the following features:

1. It models intervals of interference as periods during which the bus is not available.
2. It allows more general patterns of interferences to be specified, compared with Tindell's model.
3. It allows the combined effects of multiple sources of interference to be modeled.

The definition of $E_i(t)$ according to the generalized deterministic error model is based on the following:

- There are k sources of interference, with each source l contributing an error term $E_i^l(t)$. Their combined effect is $E_i(t) = E_i^1(t)|E_i^2(t)|\cdots|E_i^k(t)$, where | denotes composition of error terms. In this chapter addition (+) is pessimistically used to compose error terms.
- Each source l interferes by inducing an undefined bus value during a characteristic time period I^l. Each such interference will lead to a transmission error. If I^l is larger than τ_{bit}, then the error recovery will be delayed accordingly.
- Patterns of interferences for each source l can independently be specified as
  - An initial b^l groups of bursts with period T_f^l, where each group consists of n^l interferences of length I^l and with period t_f^l.
  - A residual error rate of single interferences of length I_r^l per r_f^l time units after the initial group of bursts, that is, after $b^l * T_f^l$.

Figure 13.9 illustrates the interference pattern from a single source with $b^l = 4$ and $n^l = 3$.

Now $E_i(t)$ is defined for the case of k sources of interference:

$$E_i(t) = E_i^1(t)|E_i^2(t)|\cdots|E_i^k(t) \tag{13.16}$$

where

$$\begin{aligned} E_i^l(t) = \text{Bu}^l(t) * [O_i^b + \max(0, I^l - \tau_{\text{bit}})] \\ +\text{Re}^l(t) * [O_i^r + \max(0, I_r^l - \tau_{\text{bit}})] \end{aligned} \tag{13.17}$$

where

$$\text{Bu}^l(t) = \min\left(n^l * b^l, \left\lfloor \frac{t}{T_f} \right\rfloor * n^l + \min\left(n^l, \left\lceil \frac{t \bmod T_f^l}{t_f^l} \right\rceil\right)\right) \tag{13.18}$$

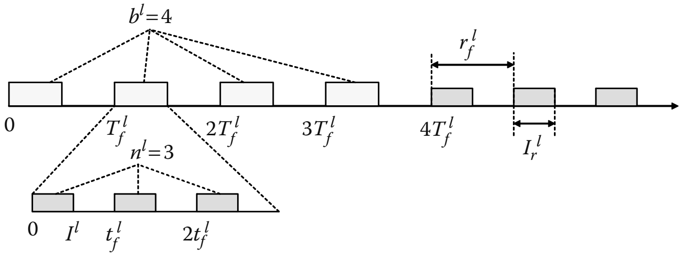


FIGURE 13.9 Interference pattern from a single source with $b^l = 4$ bursts each containing $n^l = 3$ interferences, followed by a residual error of single interferences every r_f^l time units.

and

$$\mathrm{Re}^l(t) = \max\left(0, \left\lceil \frac{t - T_f^l * b^l}{r_f^l} \right\rceil\right) \tag{13.19}$$

Some explanations are given below:

1. $\max(0, I^l - \tau_{\mathrm{bit}})$ defines the length of I^l exceeding τ_{bit}.
2. $n^l * b^l$ is the maximum number of interferences in the initial bursty period.
3. $\lfloor t/T_f \rfloor$ is the number of full bursts until t.
4. $\left\lceil \frac{t \bmod T_f^l}{t_f^l} \right\rceil$ is the number of t_f^l periods that fit in the last (not completed) burst period in t.

This model uses separate overheads for bursts (O_i^b) and single errors (O_i^r) in the above equation for $E_i^l(t)$. The advantage of giving separate overhead factors is that it will allow more accurate modeling if these factors are known. This could reduce pessimism especially in burst cases with $O_i^b > t_f^l$, since in such cases we can combine several burst errors into one error plus the burst width $(O_i^s + n^l * t_f^l)$. However, for many practical applications and during system design, one may assume the overheads for burst error and single error to be the same, denoted by O_i and given by

$$O_i = 23 * \tau_{\mathrm{bit}} + \max_{k \in \mathrm{hp}(i) \cup \{i\}} (C_k) \tag{13.20}$$

It can easily be seen that, Tindell's model is a special case of this generalized model, with one source ($k = 1$), one initial burst ($b^l = 1$) with n_{error} number of burst errors ($\mathrm{Bu}^1(t) = n_{\mathrm{error}}$), $r_f^1 = T_{\mathrm{error}}$, and the interferences $I^l = I_r^l = \tau_{\mathrm{bit}}$. However, there are no provisions for specifying durations of interference (I^l) or parameters for groups of bursts (b^l and T_f^l) in Tindell's model.

13.6.4 Probabilistic Error Models

Both the models, presented by Tindell et al. and Punnekkat et al., are based on an assumption of minimum interarrival time, that is, bounded number of errors in a specified interval. However, several sources of interference, for example, EMI, can be more accurately described as a random pulse train following specific probability distributions [45]. Trying to represent this using minimum interarrival times is not easy. Rather, a probabilistic approach would be more suitable.

Navet et al. [46] present a probabilistic error model for CAN, where the errors are described as stochastic processes. These stochastic processes consider both the frequency of the errors and their gravity. Both single-bit errors and bursts of errors can be represented.

However, as the approach presented by Navet et al. [46] is pessimistic, Broster et al. [47] present a more accurate probabilistic approach, though they do not consider burst errors. Using the approach by Broster et al., distributions of worst-case response-times can be obtained when there are probabilistic events in the system, for example, faults caused by EMI [48].

Hansson et al. [49,50] present a completely different approach. Here the schedulability of the system is determined using simulation. Using simulation even more complex sources of interference can be used, achieving a more realistic result compared to the analytic approaches described above. However, the weakness of using simulation is that it is hard to determine whether or not the coverage of the simulation is good enough for the considered application.

13.7 Holistic Analysis

In this section it is shown how to apply the CAN analysis of Section 13.5 in a distributed system comprising of "end-to-end" timing requirements. It is assumed that a distributed system consists of a number of nodes that are interconnected with a CAN bus. On the nodes, a number of tasks are executed, scheduled by a preemptive fixed-priority real-time scheduler. Timing requirements can be both on a task level, where task response-times have to be less than task deadlines, as well as end-to-end requirements where response-times are measured from the initiation of a task on one node, to the completion of a task on another node. Here, the task on the first node sends a frame, triggering the start of the task on the second node. In the following, the determination of the fulfillment of such requirements using holistic analysis is shown [51].

13.7.1 Attribute Inheritance

One of the main features of holistic analysis comes from the idea of attribute inheritance: a task is invoked by some event, and will after some or its entire execution time queue a frame. The frame could be queued soon after the event (e.g., if there were no higher priority tasks, and the execution time to queue the frame was small). The frame could be queued late after the event: when the worst-case scheduling scenario, from the sending task's point of view, occurs. This difference between the shortest and longest queuing times is the queuing jitter. For simplicity, we can use zero as the shortest queuing time. The longest time is upper bounded by the worst-case response-time of the queuing task. Hence, the queuing jitter of a frame i is given by

$$J_i = R_{\text{send}(i)} \tag{13.21}$$

where send(i) is the task that sends frame i.

Suppose that a task is released when a frame arrives. Then, this task inherits release jitter from the frame in just the same way as a frame inherits queuing jitter from the sending task. The earliest time that a frame i can reach the destination processor is 47 bit times (the smallest frame is 47 bits long). The latest a frame i can reach the destination processor is R_i. Hence, the destination task, dest(i), inherits a release jitter of

$$J_{\text{dest}(i)} = R_i - 47\tau_{\text{bit}} \tag{13.22}$$

The worst-case response-time of task dest(i), $R_{\text{dest}(i)}$, is measured from $47\tau_{\text{bit}}$ after the event invoking task send(i). So, the time $47\tau_{\text{bit}}+R_{\text{dest}(i)}$ gives the end-to-end time from the event invoking task send(i) to the completion of the task dest(i).

13.7.2 Holistic Scheduling Problem

The equations giving worst-case response-times for tasks depend on the timing attributes of frames arriving at the processor (e.g., the release jitter of a task kicked off by a frame arrival is inherited from the worst-case response-time of the frame; also overheads from "frame arrived" interrupts are dependent on the worst-case response times of incoming frames). Hence, processor scheduling analysis cannot be applied before the bus scheduling analysis. However, equations giving worst-case response-times for frames depend on the timing attributes of tasks queuing the frames (e.g., the queuing jitter of a frame queued by a task is inherited from the worst-case response-time of the frame). Hence, bus scheduling analysis cannot be applied before the processor scheduling analysis.

In conclusion, processor scheduling analysis cannot be applied before the bus scheduling analysis, and vice versa. The problem is similar to the recurrence equations in Section 13.5, where R_i appears on both sides of several equations.

The first step is to assume that all the calculated timing attributes (R_i and J_i) are zero. Then, both sets of analysis can be applied (for processors and for the bus) to obtain new values. Following this, the analysis can be repeated using the newly obtained values. The iteration carries on until the calculated values do not change anymore.

13.7.3 Example

Consider a simple distributed real-time system consisting of two nodes (nodes A and B) interconnected with a CAN. Three tasks TA_1, TA_2, and TA_3 are executing on node A and three tasks TB_1, TB_2, and TB_3 are executing on node B. The task TA_2 on node A is activated by the arrival of a CAN frame ($M_{B_1A_2}$), which is queued by the task TB_1 on node B. Symmetrically, the task TB_2 on node B is activated by the arrival of a CAN frame ($M_{A_1B_2}$), which is queued by the task TA_1 on node A. The latter frame has

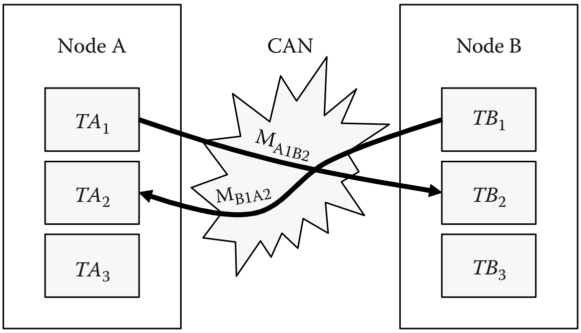


FIGURE 13.10 Example system.

TABLE 13.2 Example System Parameters

Task	Period T	Deadline D	Jitter J	Computation Time C
TA_1	100	—	3	7
TA_2	—	100	—	5
TA_3	50	25	2	10
TB_1	80	—	2	6
TB_2	—	100	—	5
TB_3	50	25	2	10

higher priority than the former, but both have higher priority than the low priority CAN frames that are also sent on the bus. The example is outlined in Figure 13.10. In Table 13.2, the deadline of tasks TA_2 and TB_2 are given relative to the release of TB_1 and TA_1, respectively.

In order to calculate end-to-end response times, three sets of coupled equations are used: one for the CAN bus and one for each of the nodes. The basic equation for the nodes is

$$R_i = J_i + w_i \tag{13.23}$$

where w_i is given by

$$w_i = C_i + B_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \tag{13.24}$$

The equation for the CAN bus is

$$R_i = J_i + w_i + C_i \tag{13.25}$$

where w_i is given by

$$w_i = B_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j + \tau_{\text{bit}}}{T_j} \right\rceil C_j \tag{13.26}$$

Following the example system in Table 13.2, these two equations are instantiated. The first thing to decide are the priority orderings among the tasks running on the two nodes, and in this example Deadline Monotonic [52] priority assignment algorithm is used (an optimal algorithm can be found in Ref. [22]). However, all deadlines are not known yet, as they rely on the temporal behavior of parts of the system. Hence, initially uninstantiated variables have to be used in some cases. Looking at the unknown deadline D_{TA_1}, TA_1 terminates by sending a CAN frame $M_{A_1B_2}$ to node B. This frame will invoke task TB_2 which has a deadline of 100 relative to the invocation of TA_1. Consequently, D_{TA_1} can be defined as follows:

$$D_{TA_1} = 100 - R_{TB_2} - R_{M_{A_1B_2}}$$

where $R_{M_{A_1B_2}}$ is the worst-case response-time for the CAN frame sent from TA_1 to TB_2.

Similarly

$$D_{TB_1} = 100 - R_{TA_2} - R_{M_{B_1A_2}}$$

where $R_{M_{B_1A_2}}$ is the worst-case response-time for the CAN frame $M_{B_1A_2}$ sent from TB_1 to TA_2.

The unknown task jitter J_{TA_2} is dependent on variations in the arrival of M_{B1A2} frames, that is,

$$J_{TA_2} = R_{M_{B_1A_2}} - 47\tau_{\text{bit}}$$

where $47\tau_{\text{bit}}$ is the minimum and $R_{M_{B_1A_2}}$ the maximum frame response-time (relative to the release of TB_1).

Similarly

$$J_{TB_2} = R_{M_{A_1B_2}} - 47\tau_{\text{bit}}$$

The only remaining unknowns in Table 13.2 are the periods for tasks TA_2 and TB_2. As they are both kicked off by tasks (TB_1 and TA_1) with known periods, they will inherit the periods of these tasks, that is, $T_{TA_2} = 80$ and $T_{TB_2} = 100$.

Using the above defined parameters, the schedulability equations are defined as follows, starting with node A:

$$\begin{aligned}
R_{TA_1} &= 3 + w_{TA_1} \\
w_{TA_1} &= 7 + B_{TA_1} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \\
B_{TA_n} &= 0 \\
J_{TA_2} &= R_{M_{B_1A_2}} - 47\tau_{\text{bit}} \\
R_{TA_2} &= J_{TA_2} + w_{TA_2} \\
w_{TA_2} &= 5 + B_{TA_2} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \\
R_{TA_3} &= 2 + w_{TA_3} \\
w_{TA_3} &= 10 + B_{TA_3} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j
\end{aligned}$$

The equations for the tasks on node B are symmetrical:

$$\begin{aligned}
R_{TB_1} &= 2 + w_{TB_1} \\
w_{TB_1} &= 6 + B_{TB_1} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \\
B_{TB_n} &= 0 \\
J_{TB_2} &= R_{M_{A_1B_2}} - 47\tau_{\text{bit}} \\
R_{TB_2} &= J_{TB_2} + w_{TB_2} \\
w_{TB_2} &= 5 + B_{TB_2} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j
\end{aligned}$$

$$R_{TB_3} = 2 + w_{TB_3}$$

$$w_{TB_3} = 10 + B_{TB_3} + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j$$

Remains to formulate the equations for the CAN-bus:

$$R_{M_{A_1B_2}} = R_{TA_1} + w_{M_{A_1B_2}} + 135\tau_{\text{bit}}$$

$$w_{M_{A_1B_2}} = 135\tau_{\text{bit}}$$

$$R_{M_{B_1A_2}} = R_{TB_1} + w_{M_{B_1A_2}} + 135\tau_{\text{bit}}$$

$$w_{M_{B_1A_2}} = 135\tau_{\text{bit}} + \left\lceil \frac{w_{M_{B_1A_2}} + R_{TA_1}}{100} \right\rceil 135\tau_{\text{bit}}$$

The above are a set of mutually dependent equations that can be solved using iteration, that is, by initially setting all variables (R_{TA_1}, R_{TA_2}, R_{TA_3}, R_{TB_1}, R_{TB_2}, R_{TB_3}, $R_{M_{A_1B_2}}$, and $R_{M_{B_1A_2}}$) to zero and then iteratively use these values in the equation to obtain new variable values. The iteration stops either when all new values are equal to the previous values (which means that the system is found schedulable), or if any of the deadlines are violated (which would mean that the system is not schedulable). Note that priorities are updated according to Deadline Monotonic [52] and hence, hp(i) might change during the iterations. More information on the holistic scheduling problem and solutions can be found in Refs. [53–55].

13.8 Middlewares and Frame Packing

The holistic analysis introduced in Section 13.7 typically requires its implementation in tools in order to calculate the schedulability of the CAN bus. However, one weak point of the "holistic system design" is that tasks trigger transmission of frames, which (once transmitted) in turn triggers task, which in turn can trigger frames, etc. Although suitable for some applications, depending on the complexity of these inter dependencies between task executions and frame transmissions, the calculations can sometimes be problematic. For this reason, it is not uncommon to use a middleware technology for removing the task-frame interdependencies.

When building distributed automotive applications, the applications send and receive elementary pieces of information, called signals. When an application is reading a signal, it might be produced at a remote location, reachable over the network. Using a middleware, this can be made transparent to the signal's reader. Typically, the signals have a limited lifetime in the sense that its value becomes useless some time after it has been produced, that is, the signal has requirements on data freshness. It is not uncommon that signals have a very small size, for example, one bit value. Hence, due to stringent requirements on low resource usage, multiple signals are packed into one CAN frame. Then, frame-packing algorithms are used in conjunction with the response-time analysis presented in Section 13.5 in order to find proper periods and priorities of these frames, satisfying the data freshness requirements of the signals. These algorithms are implemented in tools that in conjunction with a proper middleware technology provide automotive applications with fresh signal data and minimized bandwidth consumption. An example of such a tool and middleware, used by, for example, Volvo, is the Volcano concept* [56–59]. More detailed information on how frame packing is done can be found in Refs. [60,61]. A middleware expected to be used in large scale is under development in the AUTOSAR project† (see Chapter 2), and by the usage of a proper configurations tool, real-time guarantees are supposed to be possible. Other middlewares, that do not provide real-time guarantees, include J1939 [6] and CANopen [32].

13.9 Summary

This chapter has presented the CAN, which is the predominant communication standard within the automotive sector. CAN is a collision-free broadcast-priority-based fieldbus, allowing for cheap and robust implementations of small-scale networks in control applications. CAN allows for the implementation of such applications using a network with a physical length of up to some 100 m, data-rates of up to 1 Mbps, and small data frames of maximum 8 bytes of payload.

The chapter has described the CAN protocol with frame identifiers that also act as priorities. Other characteristics of CAN presented include the frame arbitration mechanism, the error handling mechanism, and the bit-stuffing mechanism. These mechanisms, together, give a predictable, reliable, and efficient protocol suitable for a wide range of embedded systems.

In addition to describing the basic CAN protocol, we have described alternative ways to schedule frames on the CAN bus. This is typically done by software protocol implementations on top of the CAN protocol. These alternatives can be used to give CAN new sets of properties such as timing determinism and fair allocation of bandwidth among applications.

We have also shown how to calculate bounds on the delays a frame may experience before it arrives at the destination (i.e., how to calculate the worst-case response-time of a frame). The calculation methods are different for the different scheduling methods, as are the resulting response-times. Hence, in order to adapt CAN to the application at hand, it is important to select the most suitable scheduler.

We have outlined a method to calculate end-to-end response-times for chains of processing on nodes intermixed with frames sent on the bus (for instance the process of reading a sensor, sending the sensor value on the bus, and finally writing an output to an actuator). This method, called holistic analysis, can be used to calculate response-time over a whole distributed system, thus making sure that deadlines are met for distributed applications.

Finally, we have introduced how frame packing together with middleware technologies can be used to implement distributed automotive applications over CAN.

* The Volcano concept is developed by Volcano Communications Technologies AB, which was acquired by Mentor Graphics in May 2005.

† Automotive Open System Architecture (AUTOSAR), http://www.autosar.org/.

References

1. ISO 11898. Road vehicles—Interchange of digital information—Controller Area Network (CAN) for high-speed communication. International Standards Organisation (ISO), ISO Standard-11898, November 1993.
2. ISO 11519-2. Road vehicles—Low-speed serial data communication—Part 2: Low-speed Controller Area Network (CAN). International Standards Organisation (ISO), ISO Standard-11519-2, 1994.
3. SAE J1850 Standard—The Society of Automotive Engineers (SAE) Vehicle Network for Multiplexing and Data Communications Standards Committee. Class B Data Communications Network Interface, May 1994, SAE.
4. C. A. Lupini. Multiplex bus progression 2003. SAE Technical Paper Series, 2003-01-0111, 2003.
5. SAE J2284 Standard. High Speed CAN (HSC) for Vehicle Applications at 500 kbps. *SAE Standards*, 1999, SAE.
6. SAE J1939 Standard—The Society of Automotive Engineers (SAE) Truck and Bus Control and Communications Subcommittee. SAE J1939 Standards Collection, 2004, SAE.
7. ISO 11783-1. Tractors and machinery for agriculture and forestry—Serial control and communications data network—Part 1: General standard for mobile data communication. International Standards Organisation (ISO), ISO Standard-11783-1, 2005.
8. ISO 11783-2. Tractors and machinery for agriculture and forestry—Serial control and communications data network—Part 2: Physical layer. International Standards Organisation (ISO), ISO Standard-11783-2, 2002.
9. NMEA 2000^{ő} Standard. The National Marine Electronics Association. http://www.nmea.org.
10. L. Kleinrock and F. A. Tobagi. Packet switching in radio channels. Part I. Carrier sense multiple access models and their throughput-delay characteristic. *IEEE Transactions on Communications*, 23(12):1400–1416, December 1975.
11. M. Barranco, L. Almeida, and J. Proenza. ReCANcentrate: A replicated star topology for CAN networks. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'05)*, Vol. 2, Catania, Italy, September 2005, pp. 469–476.
12. M. Barranco, G. Rodríguez-Navas, J. Proenza, and L. Almeida. CANcentrate: An active star topology for CAN networks. In *Proceedings of the 5th IEEE International Workshop on Factory Communication Systems (WFCS'04)*, Vienna, Austria, September 2004, pp. 219–228.
13. J. Sommer and R. Blind. Optimized resource dimensioning in an embedded CAN–CAN gateway. In *International Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, July 2007, pp. 55–62.
14. J. Leohold. Automotive system architecture. In *Proceedings of the Summer School "Architectural Paradigms for Dependable Embedded Systems,"* Vienna University of Technology, Vienna, Austria, September 2005. pp. 545–591.
15. LIN Consortium. LIN Protocol Specification, Revision 1.3, December 2002. http://www.lin-subbus.org/.

16. LIN Consortium. LIN Protocol Specification, Revision 2.0, September 2003. http://www.lin-subbus.org/.
17. K. W. Tindell, A. Burns, and A. J. Wellings. Calculating Controller Area Network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, August 1995.
18. J. Charzinski. Performance of the error detection mechanisms in CAN. In *Proceedings of the 1st International CAN Conference*, Mainz, Germany, 1994, pp. 1–29.
19. CAN Specification 2.0, Part-A and Part-B. CAN in Automation (CiA), Am Weichselgarten 26, D-91058 Erlangen. http://www.can-cia.de/, 2002.
20. E. Tran. Multi-bit error vulnerabilities in the Controller Area Network protocol. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, 1999.
21. M. Paulitsch, J. Morris, B. Hall, K. Driscoll, E. Latronico, and P. Koopman. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, June 2005, pp. 346–355.
22. R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, April 2007.
23. J. D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, December 1983.
24. H. Zimmermann. OSI reference model: The ISO model of architecture for open system interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.
25. ISO 11898-4. Road vehicles—Controller Area Network (CAN)—Part 4: Time-triggered communication. International Standards Organisation (ISO), ISO Standard-11898-4, December 2000.
26. L. Almeida, J. A. Fonseca, and P. Fonseca. A flexible time-triggered communication system based on the Controller Area Network: Experimental results. In D. Dietrich, P. Neumann, and H. Schweinzer, editors, *Fieldbus Technology—Systems Integration, Networking, and Engineering—Proceedings of the Fieldbus Conference FeT'99*, Magdeburg, Federal Republic of Germany, September 23–24, 1999, Springer-Verlag, 1999, pp. 342–350.
27. L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Transaction on Industrial Electronics*, 49(6):1189–1201, December 2002.
28. M. Di Natale. Scheduling the CAN bus with earliest deadline techniques. In *Proceedings of the 21st IEEE International Real-Time Systems Symposium (RTSS'00)*, Orlando, FL, November 2000, pp. 259–268.
29. M. A. Livani and J. Kaiser. EDF consensus on CAN bus access for dynamic real-time applications. In J. Rolim, editor, *Proceedings of the IPPS/SPDP Workshops*, volume Lecture Notes in Computer Science (LNCS-1388), Orlando, FL, Springer-Verlag, March 1998, pp. 1088–1097.
30. M. A. Livani, J. Kaiser, and W. J. Jia. Scheduling hard and soft real-time communication in the Controller Area Network (CAN). In L.-C. Zhang, A. H. Frigeri, and W. A. Halang, editors, *Real Time Programming 1998—Proceedings of the 23rd IFAC/IFIP Workshop on Real-Time Programming*, volume IFAC Proceedings Volumes, Shantou, ROC, June 1998.

31. K. M. Zuberi and K. G. Shin. Non-preemptive scheduling of messages on Controller Area Network for real-time control applications. In *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium (RTAS'95)*, Chicago, IL, May 1995, pp. 240–249.
32. CiA. CANopen communication profile for industrial systems, based on CAL. CiA Draft Standard 301, rev 3.0, October, 1996. http://www.canopen.org.
33. SAE J1938 Standard. Design/Process Checklist for Vehicle Electronic Systems. SAE Standards, May 1998, SAE.
34. G. Cena and A. Valenzano. An improved CAN fieldbus for industrial applications. *IEEE Transaction on Industrial Electronics*, 44(4):553–564, August 1997.
35. L. Lo Bello and O. Mirabella. Randomization-based approaches for dynamic priority scheduling of aperiodic messages on a CAN network. In *LCTES '00: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, London, United Kingdom, 2001, pp. 1–18.
36. T. Nolte. Share-driven scheduling of embedded networks. PhD thesis, Department of Computer and Science and Electronics, Mälardalen University, Sweden, May 2006.
37. T. Nolte and K.-J. Lin. Distributed real-time system design using CBS-based end-to-end scheduling. In *Proceedings of the 9th IEEE International Conference on Parallel and Distributed Systems (ICPADS'02)*, Taipei, Taiwan, ROC, December 2002, pages 355–360.
38. T. Nolte, M. Nolin, and H. Hansson. Real-time server-based communication for CAN. *IEEE Transactions on Industrial Informatics*, 1(3):192–201, August 2005.
39. N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
40. K. W. Tindell and A. Burns. Guaranteed message latencies for distributed safety-critical hard real-time control networks. Technical Report YCS 229, Department of Computer Science, University of York, York, England, June 1994.
41. K. W. Tindell, H. Hansson, and A. J. Wellings. Analysing real-time communications: Controller Area Network (CAN). In *Proceedings of 15th IEEE International Real-Time Systems Symposium (RTSS'94)*, San Juan, Puerto Rico, December 1994, pp. 259–263.
42. J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE International Real-Time Systems Symposium (RTSS'90)*, Lake Buena Vista, FL, December 1990, pp. 201–209.
43. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, January 1973.
44. S. Punnekkat, H. Hansson, and C. Norström. Response time analysis under errors for CAN. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, Washington DC, May–June 2000, pp. 258–265.
45. M. J. Buckingham. *Noise in Electronic Devices and Systems*. Ellis Horwood Series in Electrical and Electronic Engineering, Ellis Horwood Ltd, London, United Kingdom, December 1983.
46. N. Navet, Y.-Q. Song, and F. Simonot. Worst-case deadline failure probability in real-time applications distributed over Controller Area Network. *Journal of Systems Architecture*, 46(7):607–617, April 2000.

47. I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium (RTSS'02)*, Austin, TX, December 2002, pp. 269–278.
48. I. Broster, A. Burns, and G. Rodríguez-Navas. Timing analysis of real-time communication under electromagnetic interference. *Real-Time Systems*, 30(1–2):55–81, May 2005.
49. H. Hansson, T. Nolte, C. Norström, and S. Punnekkat. Integrating reliability and timing analysis of CAN-based systems. *IEEE Transaction on Industrial Electronics*, 49(6):1240–1250, December 2002.
50. H. Hansson, C. Norström, and S. Punnekkat. Integrating reliability and timing analysis of CAN-based systems. In *Proceedings of the 3rd IEEE International Workshop on Factory Communication Systems (WFCS'00)*, Porto, Portugal, September 2000, pp. 165–172.
51. K. W. Tindell and J. Clark. Holistic schedulability analysis for distributed real-time systems. *Microprocessing and Microprogramming—Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.
52. J.-T. Leung and J. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.
53. I. Bate and A. Burns. Investigation of the pessimism in distributed systems timing analysis. In *Proceedings of the 10th Euromicro Conference on Real-Time Systems (ECRTS'98)*, Berlin, Germany, June 1998, pp. 107–114.
54. W. Henderson, D. Kendall, and A. Robson. Improving the accuracy of scheduling analysis applied todistributed systems computing minimal responsetimes and reducing jitter. *Real-Time Systems*, 20(1):5–25, 2001.
55. J. C. Palencia and M. González Harbour. Exploiting precedence relations in the schedulability analysis ofdistributed real-time systems. In *Proceedings of the 20th IEEE International Real-Time Systems Symposium (RTSS'99)*, Phoenix, AZ, December 1999, pp. 328–339.
56. J. Axelsson, J. Fröberg, H. Hansson, C. Norström, K. Sandström, and B. Villing. Correlating bussines needs and network architectures in automotive applications—A comparative case study. In *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and Their Applications (FET'03)*, Aveiro, Portugal, July 2003, pp. 219–228.
57. L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano—A revolution in on-board communication. Volvo Technology Report 98-12-10, 1998.
58. A. Rajnák. The LIN standard. In R. Zurawski, editor, *The Industrial Communication Technology Handbook*, CRC Press, Taylor & Francis Group, Boca Raton, FL, 2005, pp. 31-1–31-13.
59. A. Rajnák. Volcano: Enabling correctness by design. In R. Zurawski, editor, *The Industrial Communication Technology Handbook*, CRC Press, Taylor & Francis Group, Boca Raton, FL, 2005, pp. 32-1–32-18.
60. R. Saket and N. Navet. Frame packing algorithms for automotive applications. *Journal of Embedded Computing (JEC)*, 2:93–102, 2006.
61. K. Sandström, C. Norström, and M. Ahlmark. Frame packing in real-time communication. In *The 7th International Workshop on Real-Time Computing Systems and Applications (RTCSA'00)*, Cheju Island, South Korea, December 2000.

14

Scheduling Messages with Offsets on Controller Area Network: A Major Performance Boost

Mathieu Grenier
Lorraine Laboratory of Computer Science Research and Applications

Lionel Havet
National Institute for Research in Computer Science and Control

Nicolas Navet
National Institute for Research in Computer Science and Control

14.1 Introduction

Controller area network (CAN) has been and will most likely remain a prominent network in passenger cars for at least two more car generations. One of the issues CAN will have to face is the growth of traffic with the increasing amount of data exchanged between electronic control units (ECUs). A car manufacturer has to make sure that the set of frames will be schedulable, that is, the response time of the frames is kept small enough to ensure that the freshness of the data is still acceptable when used at the receiver end. Clearly here, for most messages, even periodic ones, we are in the realm of soft real-time constraints: a deadline constraint can be occasionally missed without major consequences. However, the issue on CAN is that worst-case response times (WCRT) increase drastically with the load, which may explain why currently the bus utilization is typically kept at low levels (up to 30% or 40%) and why FlexRay is considered as a must for next generation architectures.

Scheduling theory (see, for instance, Ref. [1]) tells us that the WCRT for a frame corresponds to the scenario where all higher priority CAN messages are released synchronously. Avoiding this situation, and thus reducing WCRT, can be achieved by scheduling a stream of messages with offsets. Precisely, the first instance of a stream of periodic frames is released with a delay, called the offset, with regard to a reference point which is the first time at which the station is ready to transmit. Subsequent frames of the streams are then sent periodically, with the first transmission as time origin. The choice made for the offset values has an influence on the WCRT, and the challenge is to set the offsets in such a way as to minimize the WCRT, which involves spreading the workload over time as much as possible.

Assigning offsets is a problem that has been addressed in Refs. [2,3] concerning the preemptive scheduling of tasks. It turns out that these solutions are not efficient when applied to the scheduling of messages because automotive message sets have certain specific characteristics (small number of different periods, etc.). We propose here an algorithm tailored for automotive CANs, which proved to be efficient in experiments conducted on realistic message sets generated with NETCARBENCH [4]. Then, we study the extent to which offsets can be beneficial in terms of schedulability and how they can help to better cope with higher network loads. In addition, the chapter provides some insight into the fundamental reasons why offsets are so efficient, which may lead to further improvements.

Section 14.2 discusses the algorithm we propose to assign offsets. Section 14.3 describes the experimental setup. The improvements brought by offsets in terms of response times are studied in Section 14.4. Finally, Section 14.5 studies the extent to which offsets enable dealing with higher network loads.

14.2 Offset Assignment Algorithm

The problem of best choosing the offsets has been shown in Ref. [2] to have a complexity that grows exponentially with the periods of the tasks and there is no known optimal solution that can be used in practical cases. However, there are heuristics with a low complexity, see Refs. [2,3]. In our experiments, if these algorithms are effective for task scheduling, they are not well suited to message scheduling in the automotive context, which motivates the design of a new offset assignment algorithm.

With no additional protocol, there is no global synchronization among the stations in a CAN, which means that each station possesses its own local time and that the desynchronizations between the streams of frames are local to each station. This implies that there is always the possibility that frames of any two streams coming from distinct stations are released at the same time, inducing delays for some frames. If one wants to implement a global synchronization among the ECUs, in addition to the complexity and the overhead of the clock synchronization algorithm (see, for instance, Ref. [5]), the cases of ECU reboots and local clocks that are drifting apart should be dealt with in order to obtain a robust mechanism. This certainly could be done, for instance, by building on the experience gathered with TTCAN, but at the expense of some additional complexity in the communication layers.

In this study, the offset assignment algorithm is executed on each station independently. The underlying idea of the algorithm is to distribute the workload as uniformly as possible over time, in order to avoid synchronous releases leading to traffic peaks and thus to large frame response times. More precisely, we will try to schedule the transmissions as far apart as possible.

14.2.1 Design Hypotheses and Notations

The algorithm makes the following hypotheses, which are in our experience most often met in the automotive context:

1. There are only a few distinct values for the periods (e.g., 5–10). The algorithm proposed in this study has been conceived to take advantage of this property and its efficiency relies on it. The cases with many different period values can be treated efficiently with the algorithms proposed in Refs. [2,3].
2. The time is discrete with a certain granularity: the offsets of the streams, and their periods, are multiples of g, the period of the communication task in charge of issuing the transmission requests to the communication controller. Typically, g is smaller than 5 ms.

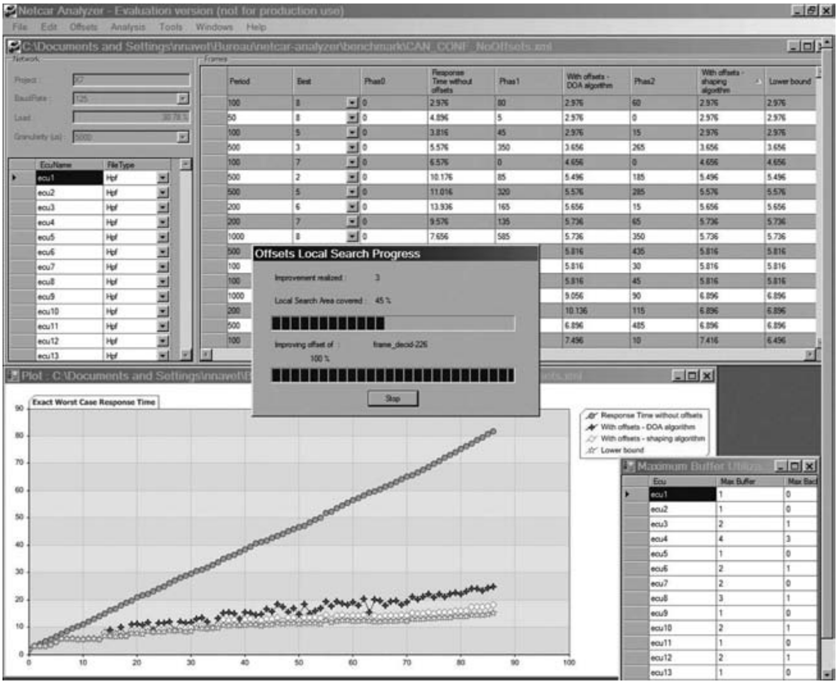
DEFINITION 14.1 *A time instant that is a multiple of g is called a* ***possible release time****. By definition, the ith possible release time, with $i \in \mathbb{N}^+$, occurs at time $(i-1)\cdot g$.*

14.2.2 Notations

On station i, the kth stream of frames, denoted by f_k^i, is characterized by the tuple $(C_k^i, D_k^i, T_k^i, O_k^i,)$: each frame produced by the stream has a worst-case transmission time equal to C_k^i, a relative deadline D_k^i (i.e., the frame must be received 10 ms after its release), and T_k^i is the transmission period for stream f_k^i. The number of streams on station i is denoted by n^i. For the sake of clarity, it is assumed that there are no jitters on the release times of the frames but they could be taken into account in the analysis. The first release time of f_k^i on station i occurs at O_k^i, which is the offset of f_k^i. Said differently, O_k^i is the duration between the first instant at which the station is operational and the transmission of the first frame of stream f_k^i. In the following, to keep the notations as simple as possible, the index of the station will not be indicated because the algorithm is executed on each station independently without considering the streams of the other stations.

14.2.3 Tool Support for WCRT Analysis

At the time of writing, to the best of our knowledge, there is no result available in the scientific literature that allows to compute response times with offsets on large sets of messages (i.e., more than 50 messages) in reasonable time. However, some


Netcar Analyzer - Evaluation version (not for production use)
File
Edit
Offsets
Analysis
Tools
Windows
Help
C:\Documents and Settings\nnavet\Bureau\netcar-analyzer\benchmark\CAN_CONF_NoOffsets.xml
Network
Frames
Plot : C:\Documents and Settings\nnavet\B
Exact Worst Case Response Time
0
10
20
30
40
50
60
70
80
90
100
Response Time without offsets
With offsets - DOA algorithm
With offsets - shaping algorithm
Lower bound
Maximum Buffer Utiliza
Stop

Project : [illegible]
BaudRate : 125
Load : 30.78 %
Granularity (us) : 5000

EcuName	FileType
ecu1	Hpf
ecu2	Hpf
ecu3	Hpf
ecu4	Hpf
ecu5	Hpf
ecu6	Hpf
ecu7	Hpf
ecu8	Hpf
ecu9	Hpf
ecu10	Hpf
ecu11	Hpf
ecu12	Hpf
ecu13	Hpf

Period	Best	Phas0	Response Time without offsets	Phas1	With offsets - DOA algorithm	Phas2	With offsets - shaping algorithm	Lower bound
100	8	0	2.976	80	2.976	60	2.976	2.976
50	8	0	4.896	5	2.976	0	2.976	2.976
100	5	0	3.816	45	2.976	15	2.976	2.976
500	3	0	5.576	350	3.656	265	3.656	3.656
100	7	0	6.576	0	4.656	0	4.656	4.656
500	2	0	10.176	85	5.496	185	5.496	5.496
500	5	0	11.016	320	5.576	285	5.576	5.576
200	6	0	13.936	165	5.656	15	5.656	5.656
200	7	0	9.576	135	5.736	65	5.736	5.736
1000	8	0	7.656	585	5.736	350	5.736	5.736
500					5.816	435	5.816	5.816
100					5.816	30	5.816	5.816
100					5.816	45	5.816	5.816
1000					9.056	90	6.896	6.896
200					10.136	115	6.896	6.896
500					6.896	485	6.896	6.896
100					7.496	10	7.416	6.496

Offsets Local Search Progress

Improvement realized : 3

Local Search Area covered : 45 %

Improving offset of : frame_decid-226

100 %

Ecu	Max Buffer	Max Bacl
ecu1	1	0
ecu2	1	0
ecu3	2	1
ecu4	4	3
ecu5	1	0
ecu6	2	1
ecu7	2	0
ecu8	3	1
ecu9	1	0
ecu10	2	1
ecu11	1	0
ecu12	2	1
ecu13	1	0

FIGURE 14.1 Screenshot of NETCAR-Analyzer during an optimization run. The left-hand graphic shows the response times (by decreasing priority) for different offset configurations. The spreadsheet in the background contains the set of frames, the different offset configurations tested, the corresponding WCRT, and certain characteristics of the ECUs, such as the queuing policy at the microcontroller level (e.g., FIFO or prioritized).

commercial products offer this feature, which is actually needed by car manufacturers. In this study, the WCRT of the frames are computed with the software NETCAR-Analyzer, first developed at INRIA, then taken over by the company RealTime-at-Work, which implements exact and very fast WCRT on CAN with offsets. This software also includes a set of proprietary offset assignment algorithms, fine-tuned with the experience gained in industrial use, that significantly outperform the algorithm proposed here, but they cannot be disclosed because of confidentiality. However, as it will be demonstrated, the algorithm shown here is efficient, and it constitutes a sound basis that can be improved and extended according to the user's need. For instance, as permitted by NETCAR-Analyzer, the user may want to optimize the WCRT for only a particular subset of messages, possibly according to a user-defined cost function. Figure 14.1 shows a screenshot of NETCAR-Analyzer.

14.2.4 Description of the Algorithm

Without loss of generality, the choice of the offset for stream f_k is made in the interval $[0, T_k[$. Indeed, because of the periodic nature of the scheduling (see Ref. [2] for more details), an offset $O_k \geq T_k$ is equivalent to $O_k T_k$. Once the initial offset O_k has been decided, all subsequent release times of stream f_k are set: they occur at times $O_k + i \cdot T_k$ with $i \in N$.

To spread the traffic over time, the offset of each stream f_k is chosen such that the release of its first frame, $f_{k,1}$, is "as far as possible" from other frames already scheduled. This is achieved by (1) identifying the longest interval with the smallest workload and (2) setting the offset for f_k in the middle of this interval.

14.2.4.1 Data Structure

Since for each stream f_k, the offset is chosen in the interval $[0, T_k[$, we choose to assign the offsets based on an analysis performed over time interval $[0, T_{\max}[$, where $T_{\max} = \max_{1 \leq k \leq n}\{T_k\}$.

The release times of the frames in the interval $[0, T_{\max}[$ are stored in an array R having $T_{\max}/g$ elements where the ith element $R[i]$ is the set of frames released at *possible release time* i (i.e., at time $(i-1) \cdot g$). Table 14.1 presents the release array R for the frames corresponding to the set of traffic streams $\mathcal{T} = \{f_1, f_2, f_3\}$, where $f_1 = (T_1 = 10, O_1 = 4)$, $f_2 = (20, 8)$, and $f_3 = (20, 18)$ ($T_{\max} = 20$) with a granularity $g = 2$.

For a given stream f_k, an interval is a set of *adjacent* possible release times.

DEFINITION 14.2 *For a stream f_k and a time granularity g, the possible release times i and i' are* ***adjacent*** *iff:*

$$\left|\left(i \bmod \frac{T_k}{g}\right) - \left(i' \bmod \frac{T_k}{g}\right)\right| = 1.$$

In the above formula, the modulo operators translate the fact that setting the offset of a stream f_k at possible release i is the same as choosing the possible release time $i + u \cdot T_k/g$ with $u \in \mathbb{N}$. Table 14.2 illustrates this definition with a stream f_1 having a period $T_1 = 10$ where the time granularity g is 2.

This leads to the definition of an interval.

DEFINITION 14.3 *For a stream f_k, an interval is an ordered set of possible release times where the ith and $(i+1)$th elements are adjacent. The length of this interval is the number of elements in the ordered set.*

TABLE 14.1 Release Array R of the Frames Corresponding to the Set of Traffic Streams $\mathcal{T} = \{f_1, f_2, f_3\}$ Where $f_1 = (T_1 = 10, O_1 = 4)$, $f_2 = (20, 8)$, and $f_3 = (20, 18)$ on the Interval $[0, 20[$

Time	0	2	4	6	8	10	12	14	16	18
Possible release time i	1	2	3	4	5	6	7	8	9	10
$R[i]$ (set of frames released)			$f_{1,1}$		$f_{2,1}$			$f_{1,2}$		$f_{3,1}$

The granularity g is equal to 2. The ith element $R[i]$ is the set of frames released at possible release time i. For instance, $R[3] = \{f_{1,1}\}$.

TABLE 14.2 Possible Release Times That Are Adjacent, in the Case of Stream f_1 Having a Period Equal to 10

Time	0	2	4	6	8
Possible release time i	1	2	3	4	5
Possible release times adjacent to i	{5,2}	{1,3}	{2,4}	{3,5}	{4,1}

For example, possible release times 4 and 1 are adjacent to 5.

For instance, for the stream f_1 (see Table 14.2), the set $\{4, 5, 1, 2\}$ is an interval of adjacent possible release times. In the algorithm presented here, we consider only the intervals made of possible release times with the same load.

DEFINITION 14.4 *The load of possible release time i is the number of releases scheduled for transmission at i, i.e., at clock time* $(i - 1) \cdot g$.

For instance, in the example of Table 14.1, the load of possible release time 3 is 1. We denote by l_k the smallest load in the interval $[0, T_k[$, *the least loaded intervals* only comprise possible release times having a load equal to l_k. For example, in Table 14.1, l_3 is equal to 0 and interval $\{10, 12\}$ belongs to the set of the least loaded intervals in $[0, 20[$.

14.2.4.2 Description of the Algorithm

We assume that the streams are sorted by increasing the value of their period, i.e., $k < h$ implies $T_k \le T_h$. The algorithm sets iteratively the offsets of streams, from f_1 to f_n. Let us consider that the stream under analysis is f_k.

1. Set offset for f_k such as to maximize the distance between its first release $f_{k,1}$, and the release right before and right after $f_{k,1}$. Concretely:
   (a) Look for l_k in the interval $[0, T_k[$.
   (b) Look for one of the longest least loaded intervals in $[0, T_k[$, where ties are broken arbitrarily. The first (resp. last) possible release time of the interval is noted by B_k (resp. E_k).
   (c) Set the offset O_k in the middle of the selected interval, the corresponding possible release time is r_k.
   (d) Update the release array R to store the frames of f_k released in the interval $[0, T_{\max}[$:

$$\forall i \in \mathbb{N} \quad \text{and} \quad r_k + i \cdot \frac{T_k}{g} \le \frac{T_{\max}}{g}$$

$$\text{do } R\left[r_k + i \cdot \frac{T_k}{g}\right] = R\left[r_k + i \cdot \frac{T_k}{g}\right] \cup f_{k,i+1}$$

A straightforward implementation of the algorithm runs in $O(n \cdot \max_k\{T_k\}/g)$, which, in practice, does not raise any problem even with large sets of messages.

14.2.4.3 Application of the Algorithm

We consider our example where $\mathcal{T} = \{f_1, f_2, f_3\}$ with $f_1 = (T_1 = 10, O_1 = 4)$, $f_2 = (20, 8)$, $f_3 = (20, 18)$ and a time granularity equal to 2. First the algorithm decides the offset for f_1: $l_1 = 0$ (step 1.(a)), $B_1 = 1$ and $E_1 = 5$ (step 1.(b)), thus $r_1 = 3$ (step 1.(c)), which means that the offset of the stream is 4. Then array R is updated: $R[3] = \{f_{1,1}\}$ and $R[8] = \{f_{1,2}\}$ (step 1.(d)). For stream f_2, $l_2 = 0$, the selected interval is $\{4, 5, 6, 7\}$ thus $B_2 = 4$, $E_2 = 7$, and $r_2 = 5$ with $R[5] = \{f_{2,1}\}$. For stream f_3, $l_3 = 0$, the selected interval is $\{9, 10, 1, 2\}$ thus $B_3 = 9$, $E_3 = 2$, and $r_3 = 10$ with $R[10] = \{f_{3,1}\}$. The results of applying the algorithm are shown in Table 14.1.

14.3 Experimental Setup

In order to get a precise idea of the real benefits of using offsets, we tried to perform experiments on realistic CANs. However, because of confidentiality reasons, very little has been published concerning benchmarks. To the best of our knowledge, the only two publicly available benchmarks are the SAE [6] and the PSA benchmarks [7]. They have both been used several times in the literature but they are clearly no more realistic with regard to current in-vehicle networks (see, for instance, Ref. [8]).

To overcome the confidentiality issue that prevent us from publishing real sets of messages, we developed NETCARBENCH [4], a software that generates automotive sets of messages according to parameters defined by the user (network load, number of ECUs, distribution of the periods of the frames, etc.). NETCARBENCH is aimed at improving the assessment, the understanding, and the comparability of algorithms and tools used in the design of in-vehicle communication networks. To facilitate its diffusion, NETCARBENCH is released under the GPL license and is downloadable at: http://www.netcarbench.org.

We mostly find three types of CANs in a car today: power train, body, and chassis. In the following, we will consider body and chassis networks that exhibit rather distinct characteristics. In the experiments, except when explicitly stated, the randomly generated networks have an average load equal to 35% (with an interval of variation of 3% around the mean) and the characteristics shown in Table 14.3. The size of data payload in the frames is uniformly distributed between 1 and 8 bytes. There will be two types of experiments: some will focus on a particular network, while others will involve collecting statistics on a large number of networks (i.e., 1000 in the following). For the

TABLE 14.3 Configuration of the Networks Considered in the Experiments

Network	#ECUs	#Messages (stddev)	Bandwidth (kbps)	Frame Periods
Body	15–20	71 (8.5)	125	50 ms to 2 s
Chassis	5–15	58.5 (7.7)	500	10 ms to 1 s

For both body and chassis networks, the average load is 35% and the size of the data payload is drawn at random (uniform law) between 1 and 8. The periods are uniformly chosen in the set $\{50, 100, 200, 500, 1000, 2000\}$ for body networks, and in the set $\{10, 20, 50, 100, 200, 1000\}$ for chassis networks.

former type of experiments, the same body network and the same chassis network have been used throughout all the experiments.

In practice, it is often the case that a single station generates a large part of the global network load. For instance, in the body network, there is usually a station that serves as gateway to other networks, and which is responsible for a large fraction of the total load. We model that with a single station that generates about 30% of the total load. In the following, it will be mentioned explicitly when this "load concentration" configuration is used.

14.4 Benefits of Using Offsets on WCRTs

We first evaluate the performance gain with offsets in Section 14.4.1; then, in Section 14.4.2, we provide elements to explain the effectiveness of using offsets.

14.4.1 WCRT Comparison with and without Offsets

The main benefit of offsets is the reduction of the WCRT for low priority messages. Figure 14.2 shows the WCRT of the frames of a typical CAN body with and without offsets. Two offset strategies are tested: the algorithm presented in Section 14.2 and a purely random allocation. For this latter strategy, the results in Figure 14.2 are the average values over 100 random allocations. Also shown in Figure 14.2 is a lower bound on the WCRT that is provided by NETCAR-Analyzer; this bound cannot necessarily be reached in practice but is informative anyway about how good the offset allocation is. As can be seen, the WCRT is improved for all frames for which a gain is possible. The improvements become more and more pronounced as the priority decreases. For the lowest priority frame of this example, the WCRT with offsets is decreased by 43.2 ms (from 64.8 to 21.6 ms), which represents a reduction of a factor 3, compared to results without offsets. The gain is thus very large.

In the next experiments, we evaluate the performance of offset assignments over 1000 random sets of messages. The performance metric is the ratio of WCRT reduction when using offsets with the algorithm of Section 14.2. We consider body networks and chassis networks, with and without *load concentration*, that is, one station that is more loaded than the others—here this loaded station generates about 30% of the total network load. Figure 14.3 shows the distribution of the WCRT reduction ratio for the lowest priority frame without load concentration, while Figure 14.4 presents the case with load concentration.

Whatever the experimental condition, the gain is very significant, except for a few outliers out of the 4000 sets of messages that have been considered. This suggests that in practice offsets will most often be very beneficial. It can be observed that the gain is more important for chassis networks. The explanation lies probably in the fact that chassis networks comprise fewer stations than body networks, and thus the desynchronization between streams, which is purely local to the stations, is more efficient. When a single ECU generates a large fraction of the load (i.e., load concentration), the results are very similar to the case where the load is uniformly distributed over the stations, while intuitively they should be better. As suggested by Figure 14.2, at

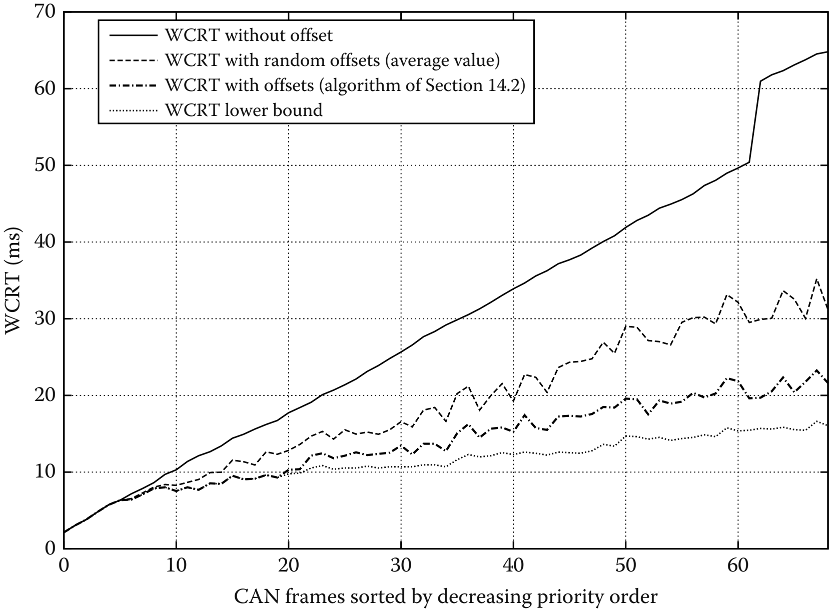


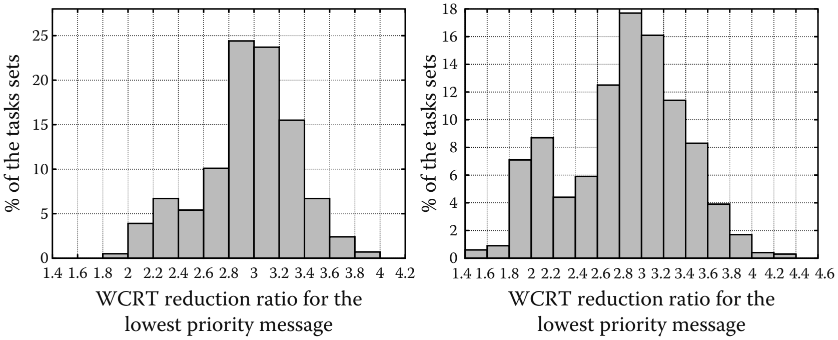
FIGURE 14.2 WCRT of the CAN frames with and without offsets for a typical 125 kbps body network with a network load of 37.6% and 68 messages. The upper curve is the WCRT without offsets, the immediate lower curve is the average value over 100 random offset allocations, the next curve is the WCRT with the algorithm of Section 14.2. Finally, the lowest curve is a lower bound on the WCRT. The steep increase of the WCRT without offsets at the end can be explained because some high priority frames have a period equal to 50, and two instances of these frames are delaying the lowest priority frames with a WCRT larger than 50 ms.

this level of load, the performance of the shaping algorithm is close to the optimal, which may explain why no difference is observed.

14.4.2 Explanation of the Gain: The Network Load Is Better Distributed

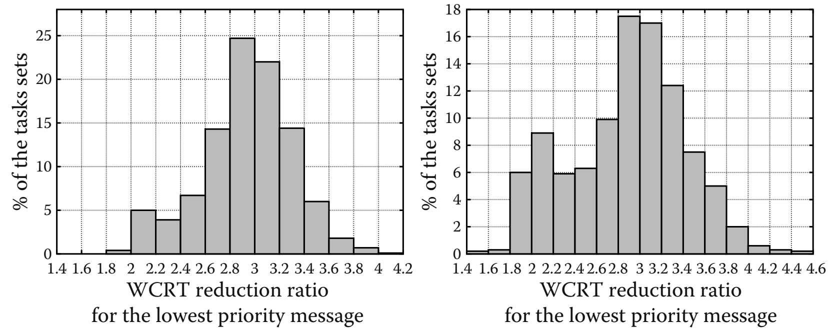
The evolution of total workload awaiting transmission (or *backlog*) is measured during 1 s (half of the 1 cm value here) with and without offsets. More precisely, when there are offsets, we consider the scenario leading to the WCRT for the lowest priority frame. Without offsets, the workload measured is the one corresponding to the synchronous case, that is, the worst-case for all frames in that context. Both workloads are plotted in Figure 14.5 for a typical body network. It can be immediately noticed that the "peaks" of the workload are much smaller with offsets, which provides a clear-cut explanation about the gains observed in Section 14.4.1. Indeed, the load awaiting transmission directly translates into response times for the lowest priority frames.

The fact that the workload with offsets in Figure 14.5 is more evenly distributed could lead us to think that there is less workload with offsets, which is actually not



(a) Body network—no load concentration (b) Chassis network—no load concentration

FIGURE 14.3 Reduction ratio of the WCRT for the lowest priority frames when offsets are used. The distribution is computed over the results obtained on a sample of 1000 random body networks (left-hand graphic) and chassis networks (right-hand graphic). The network load is uniformly distributed over the ECUs (i.e., no concentration). The x-axis is the WCRT reduction ratio (bins of size: 0.2) and the y-axis is the percentage of networks having that level of gain.



(a) Body network—with load concentration (b) Chassis network—with load concentration

FIGURE 14.4 Reduction ratio of the WCRT for the lowest priority frames when offsets are used. Same settings as Figure 14.3 except that one station alone generates, on average, 30% of the total network load (i.e., load concentration). The x-axis is the WCRT reduction ratio (bins of size: 0.2) and the y-axis is the percentage of networks having that level of gain.

the case. Figure 14.6 corrects this feeling and shows the evolution of the cumulative work arrival function over time with and without offsets for the same network as in Figure 14.5. The shape of the work arrival function with offsets is much smoother and linear than without offsets, where the "stairs" of the function are larger. This figure suggests that the algorithm of Section 14.2 performs well, and also provides us with some insight into the improvements that remain achievable, knowing that the best in terms of load distribution—but not always feasible because of the stream characteristics—would be a straight line.

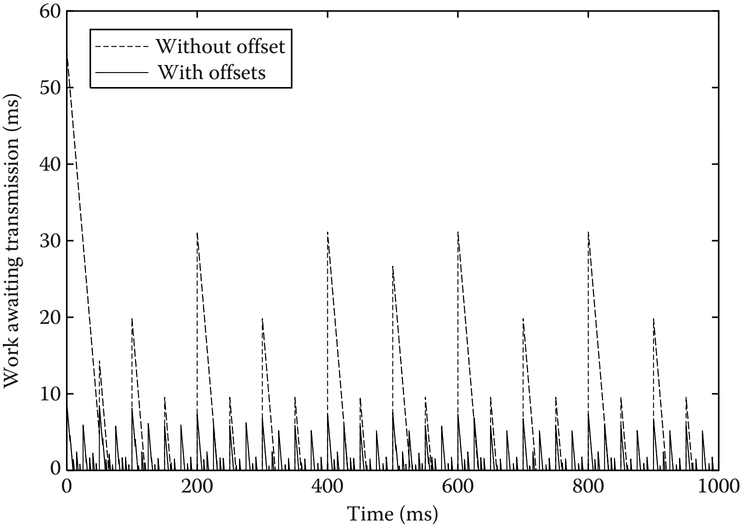


FIGURE 14.5 Amount of work awaiting transmission with and without offsets—comparison over 1 s.

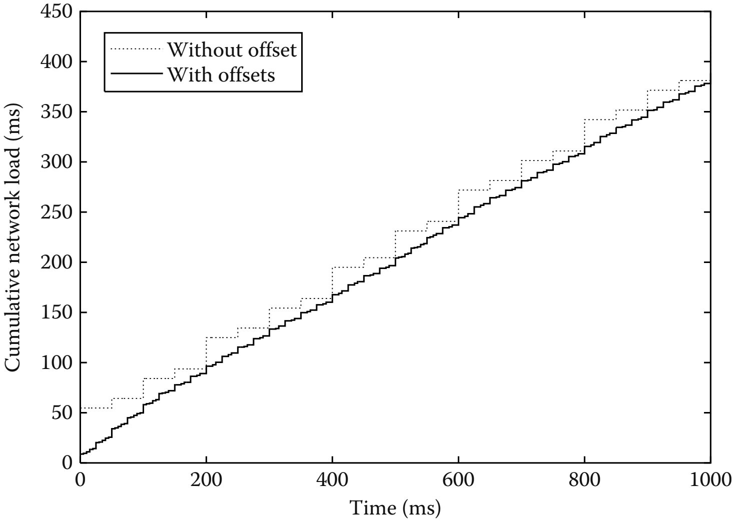


FIGURE 14.6 Cumulative network load (expressed in transmission time) with and without offsets—comparison over 1 s.

It is worth mentioning that the better distribution of the load with offsets is also very interesting for reducing peaks of CPU load since ECUs will not have to build, transmit or receive bursts of frames. In practice, this is a major reason why offsets are sometimes already implemented in production vehicles.

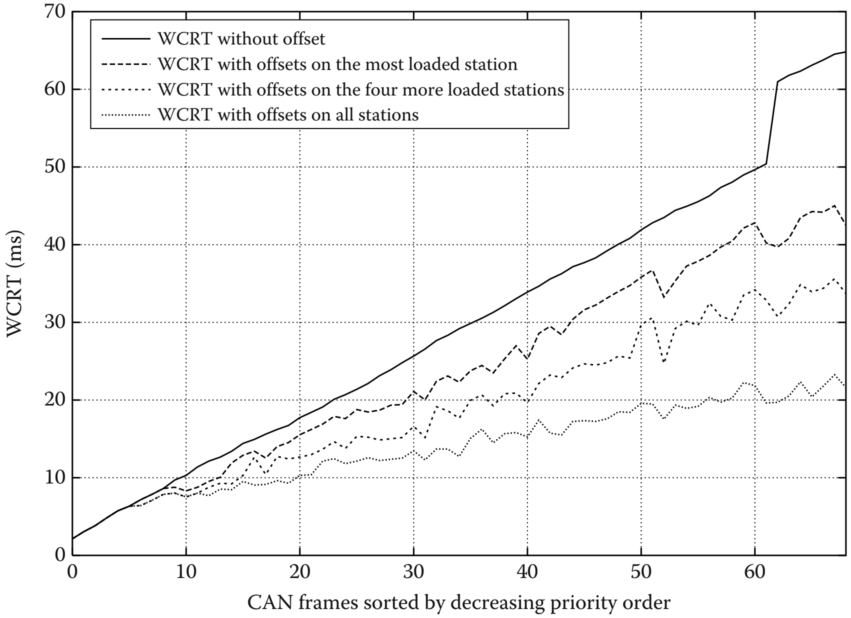


FIGURE 14.7 Comparison of the WCRT on a body network with load concentration: (1) without offsets (upper curve), (2) with offsets only on the loaded station (immediate lower curve), (3) with offsets on the four more loaded stations (third curve from the top), and (4) with offsets on all stations (lower curve).

14.4.3 Partial Offset Usage

So far, we have assumed that offset strategies would be applied to all stations. In practice, the load on a CAN is generally not evenly distributed between the stations, and it is common to have networks where a single station, or a couple of stations, induce a large fraction of the total load. In this situation, a significant improvement can already be achieved when offsets are used only on the station, or the few stations, that create most of the bus load. This also involves fewer changes for the car manufacturer.

To obtain some understanding of what to expect from offsets in this case, we generated networks where 30% of the load is concentrated on a single station (i.e., load concentration situation). Figure 14.7 shows that applying offsets on a few stations is already very advantageous in terms of WCRT of the lowest priority frame. With regard to what would be achieved without offsets, the lowest priority frame has its WCRT reduced by 34.5% with offsets on one station, and by 48% on four stations.

14.5 Offsets Allow Higher Network Loads

Up to this point, the experiments have been done on networks with a load corresponding to what is commonly found in today's automotive CANs. Now we propose to evaluate the benefits of offsets in the near future situation where network loads will increase. We model the load increase in two directions: either by distributing new messages onto existing stations or by assigning them onto new stations. We proceed as follows:

- Define a random network net_1 with a given load $load_1$. In this experiment, the body network drawn at random has a load equal to 37.6.
- Define a new load level $load_2$ (e.g., 40%). Define a random set of frames that corresponds to the load difference between $load_1$ and $load_2$. This newly created set of frames is denoted by S_{new}.
- Two methods are employed to allocate the frames of S_{new}:
  - Dispatch S_{new} on the existing stations of net_1, this new network is called net_2^{frames}
  - Dispatch the set of frames S_{new} on new stations (with a limit of five frames per station) and add them to net_1. The resulting network is called net_2^{stations}
- Determine offsets using algorithm of Section 14.2 and compute WCRT for net_2^{frames} and net_2^{stations}.

Following this procedure enables us to compare the increase of WCRT for the two scenarios identified. Figure 14.8 shows the evolution of the WCRT of the lowest priority frame for a network load ranging from 40% to 60%.

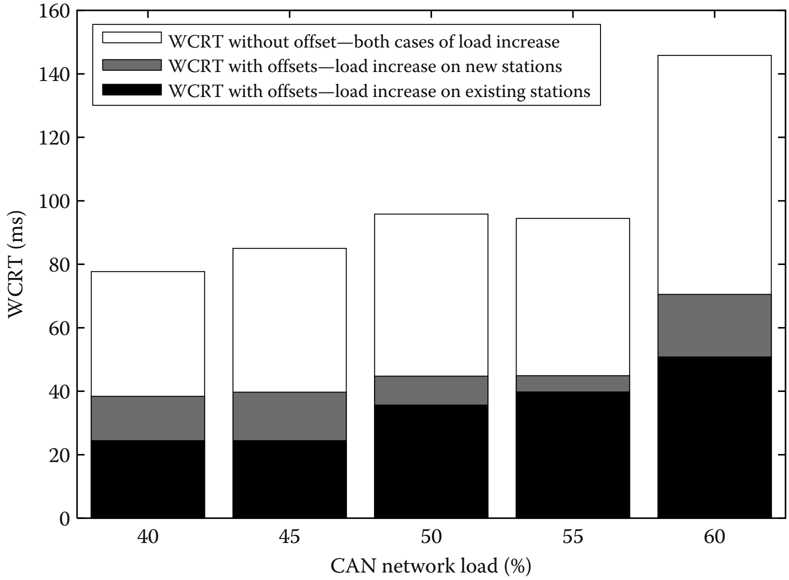


FIGURE 14.8 WCRT of the lowest priority frame for a load ranging from 40% to 60%: (1) without offsets (white), (2) with offsets and the additional load assigned to new stations (gray), and (3) with offsets and the additional load assigned to existing stations (black). The additional load is the network load added to a randomly chosen network with an initial load equal to 37.6%. The results presented here are obtained on a single typical body network.

What can be observed is that the gain with offsets remains very significant even when the load increases. For instance, at a load of 60% the gain with offsets is equal to a factor 2.8 if the additional load is distributed on existing stations, or a factor 2.1 if the additional load is allocated to new stations.

Second, the experiments show that the WCRT of the lowest priority frame with offsets at 60% is roughly similar to the WCRT at 30% of load without offsets. In other words, the performance at 60% with offsets are equivalent to the performance at 30% without offsets. Although this is not shown in Figure 14.8, this remark holds true for all frames, whatever their priority level (except at the highest priority levels where there is less gain).

Finally, it is worth noting that there is a difference whether the new load is spread over existing stations or assigned to new stations. In the latter case, offsets are less efficient in general, which is logical because the lack of global time reference implies that the offsets are local to each station.

14.6 Conclusion

This study provides two contributions. First, we propose a low-complexity algorithm for deciding offsets, which have good performances for typical automotive networks, be they body or chassis networks. This algorithm, the first of its kind in the literature to the best of our knowledge, should constitute a sound basis for further improvements and optimizations. For instance, specific constraints of a particular design process, or even vehicle project, can be taken into account.

Second, we show that the use of offsets enable very significant performance improvements on a wide range of network configurations. We believe using offsets is a robust technique that might actually provide a solution in the short term to deal with the increasing network load, and thus might allow the use of CAN as the principal network in the next car generations, at least when no safety critical functions are involved.

Offsets, which impose constraints on the frame release dates, can be seen as a trade-off between event-triggered communications and time-triggered communications. Experiments show that it is possible to achieve further gains with synchronization mechanisms between stations, which imposes additional constraints on communication and could constitute a lightweight time-triggered solution on CAN. The extent to which it can be implemented in a robust way (i.e., resilience to ECU reboots, local clocks that are drifting apart, etc.) is the subject of our ongoing work.

References

1. R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, April 2007.
2. J. Goossens. Scheduling of offset free systems. *Real-Time Systems*, 24(2):239–258, March 2003.

3. M. Grenier, J. Goossens, and N. Navet. Near-optimal fixed priority preemptive scheduling of offset free systems. In *Proceedings of the 14th International Conference on Network and Systems (RTNS'2006)*, Poitiers, France, May 30–31, 2006.
4. C. Braun, L. Havet, and N. Navet. NETCARBENCH: A benchmark for techniques and tools used in the design of automotive communication systems. In *Proceedings of the 7th IFAC International Conference on Fieldbuses and Networks in Industrial and Embedded Systems (FeT'2007)*, Toulouse, France, November 2007. Software and manual available at http://www.netcarbench.org.
5. L. Rodrigues, M. Guimaraes, and J. Rufino. Fault-tolerant clock synchronization in CAN. In *Proceedings of the 19th Real-Time Systems Symposium*, Madrid, Spain, 1998, pp. 420–429.
6. K. Tindell and A. Burns. Guaranteed message latencies for distributed safety-critical hard real-time control networks. In *First International CAN Conference Proceedings*, Germany, September 1994.
7. N. Navet, Y. Song, and F. Simonot. Worst-case deadline failure probability in real-time applications distributed over CAN (Controller Area Network). *Journal of Systems Architecture*, 46(7):407–417, 2000.
8. N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.

15

Formal Methods in the Automotive Domain: The Case of TTA

Holger Pfeifer
Ulm University

15.1 Introduction

The Time-Triggered Architecture (TTA) [1–3] is a distributed computer architecture for the implementation of highly dependable real-time systems. In particular, it targets embedded control applications, such as *by-wire* systems in the automotive or aerospace industry. For these safety-critical systems fault tolerance is of utmost importance. The Time-Triggered Protocol (TTP/C) constitutes the core of the communication level of the TTA. It furnishes a number of important services, such as atomic broadcast, consistent membership, and protection against faulty nodes that facilitate the development of these kinds of fault-tolerant real-time applications.

Formal analysis can provide an additional source of confidence in correct behavior of a system, which is particularly important in the context of safety-critical systems. Complementing the validation of applications built on top of a TTA, the fault tolerance properties of TTA itself, and its underlying communication protocol are of particular interest. Several aspects of TTP/C have therefore been formally modeled and analyzed, including clock synchronization [4], diagnostic services [5,6], the startup procedure [7–9], and the fault-tolerance properties of the central guardians [10], applying both deductive approaches based on interactive theorem proving and different kinds of model checking. The algorithms implemented in TTP/C provide challenging problems for formal analysis. This chapter describes the central mechanisms of TTP/C and their intended fault tolerance properties, and highlights the important questions that need to be addressed, and approaches to their solution, regarding both modeling and verification.

15.2 Topics of Interest

In a TTA system, a set of "nodes" are interconnected by a real-time communication system. A node consists of the host computer, which runs the application software, and the communication controller, which accomplishes the time-triggered communication between different nodes. The nodes communicate via replicated shared media, the communication "channels." There are two common physical interconnection topologies for TTA. Originally, the channels were connected to replicated passive buses, while in the more recent star topology the nodes are connected to replicated central star couplers, one for each of the two communication channels. However, the actual network topology is transparent and appears as a (logical) bus to the nodes.

The distinguishing characteristic of time-triggered systems is that all system activities are initiated by the passage of time [11]. The autonomous TTA communication system periodically executes a time-division multiple access (TDMA) schedule. Thus, access to the communication medium is divided into a series of intervals, called "slots." Every node exclusively owns certain slots in which it is allowed to send messages via the communication network. The times of the periodic message sending actions are determined *a priori*, that is, at design time of the system. The send and receive instants are contained in a message schedule, the so-called message descriptor list (MEDL). This scheduling table is static and stored at each communication controller. It thus provides common knowledge about message timing to all nodes.

The central aspects of the underlying communication protocol, viz. its fault-tolerant synchronization, diagnosis, and fault-masking algorithms and components, are described in more detail in this section.

15.2.1 Fault-Masking Functions of Central Guardians

TTA systems are designed for safety-critical applications and must therefore provide a sufficient degree of fault tolerance. The provision of fault tolerance is based on a given "fault hypothesis," that is, a set of assumptions about the types, number, and frequency of faults. The central algorithms implemented in TTP/C such as group membership and clock synchronization are able to tolerate only faults that manifest themselves as either a reception fault or a consistent send fault of some node [12], and they rely on transmission faults being consistent. That is, messages must be received correctly by either all nonfaulty nodes or none. In particular, the following three correctness properties of the TTP/C communication are required:

- *Validity:* If a correct node transmits a correct message, then all correct receivers accept the message.
- *Agreement:* If any correct node accepts a message, then all correct receivers do.
- *Authenticity:* A correct node accepts a message only if it has been sent by the scheduled sending node of the given slot.

In order to satisfy the *validity* property, for example, even faulty nodes must not send messages outside their assigned slots. In order to protect against such timing faults, special hardware components, the so-called guardians, were introduced [13]. A guardian is an autonomous unit that protects the shared communication network against faulty behavior of nodes by supervising their output. The original bus topology of the communication network employed local bus guardians, which were placed between the nodes and the bus. However, fault injection experiments showed that more sophisticated guardians are necessary to achieve the more demanding requirements for fault tolerance in the aerospace and automotive industries. In the more recent star topology, central guardians are used in the hub of each star.

By employing knowledge about the parameters of its attached nodes, guardians can judge whether a message sent by a node is valid and can be relayed to the other nodes on their channel, or whether it has to be blocked. For instance, the guardians monitor the temporal behavior of the nodes. As the time interval during which a given node is allowed to access the communication channels is statically determined, the guardians can control the correct timing of message transmissions and bar a faulty node from sending a message outside its designated slots. Thus, timing failures of nodes are effectively transformed into send faults.

Moreover, guardians can protect against a particular class of *Byzantine* faults, the so-called slightly-off-specification (SOS) faults. A component is called SOS-faulty if it exhibits only marginally faulty behavior that appears correct to some components, but faulty to others, thus violating the *agreement* property above. A slightly-off-specification timing fault could occur if the transmission of a node terminates very close to the end of its scheduled transmission interval; thus, some receivers might accept the message while others might consider it mistimed and close the reception window before the message is completely transmitted. Because the duration of a particular transmission is known beforehand, the guardian can prevent such a cut-off scenario. A node must begin its transmission during a predefined period of time after the start of its slot, otherwise the guardian would terminate the right to access the communication network. Thus, the guardian can effectively prevent cut-off SOS faults, provided that the transmission interval is chosen long enough to ensure that a transmission fits the interval whenever it is started in time. Specifically, TTP/C guardians protect against SOS faults in the line encoding of messages at the physical layer, SOS timing faults, transmission of data outside the designated sending slots, transmission of nonagreed critical state information, and against faulty nodes masquerading as other nodes, which violates the "authenticity" property above. These node failures are effectively transformed into either send or receive faults, which can be tolerated by the protocol algorithms.

15.2.2 Group Membership and Clique Avoidance

Group membership is one of the central mechanisms of the TTP/C. It provides a consistent view to all nonfaulty nodes about which nodes are operational and which are not.

The TDMA communication strategy enables a direct way to detect failures: since nodes have designated slots to send messages, the lack of a transmission is interpreted as an indication of failure. More precisely, if a node receives a correct message in a given slot it considers the respective sender operational. In the case of message omissions, a node takes an egocentric viewpoint and considers the sending node faulty rather than immediately assuming a fault of itself. Consequently, using messages only as life-signs of the sender is obviously not sufficient to cope with faulty behavior of nodes. In order to allow for nodes to diagnose faults, the messages also carry information about the sender's local perception of the current membership, which is appended to the ordinary data in each message.

Besides determining whether or not some other node is faulty, it is also necessary for a node to be able to self-diagnose a fault of its own. Nodes can inform a sender of a message of a fault by means of an acknowledgment mechanism. In TTP/C, acknowledgment is accomplished implicitly. Instead of sending a separate reply message to the original sender p, the next broadcaster q sends the acknowledgment information as part of its ordinary message, by keeping or removing p in its local membership. The previous broadcaster analyses this information to decide whether or not its original broadcast was successful. If p finds q not acknowledging the original message, either p failed to broadcast or q is receive-faulty. To resolve this ambiguity, the broadcaster following q will draw the final decision, as it will agree with either p or q.

A similar mechanism could be used for diagnosing receive faults: if a node does not receive an expected message it could check whether the next broadcaster maintained the original sender in its membership set, in which case the receiver must realise that it has suffered from a receive fault. However, TTP/C employs a slightly different mechanism that is also used to avoid the formation of disjoint cliques at the same time. A clique is a group of nodes where agreement on the current state is reached only within the group. A receiving node will always exclude the current broadcaster from its membership set if it does not agree with the membership view of the broadcaster or if no message is received at all. In addition, each node maintains two counters that keep track of how many messages the node has "accepted," that is, successfully received, and "rejected," respectively. A node will increment its counter of rejected messages if it does not agree with the broadcaster's view on the membership, while the counter of accepted messages is incremented each time the membership sets match. In its next broadcast slot the node checks whether it has accepted more messages in the last round than it has rejected. If so, the node resets the counters and broadcasts. The other case indicates that the node is in disagreement with the majority of nodes and hence will not broadcast, thus informing the other nodes about its failure.

For the group membership algorithm, one is interested in the following three correctness properties holding true:

- *Validity*: At all times, nonfaulty nodes should have all and only the nonfaulty nodes in their membership sets, while faulty nodes should have removed themselves from their sets.
- *Agreement*: At all times all nonfaulty nodes should have the same membership sets.
- *Self-diagnosis*: A node that becomes faulty should eventually diagnose its fault and remove itself from its own membership set.

15.2.3 Clock Synchronization

Distributed dependable real-time systems crucially depend on fault-tolerant clock synchronization. This is particularly true for the TTA, in which the nodes perform their actions according to a predetermined, static schedule, that is, triggered by the passage of time. Obviously, it is essential that the clocks of all nodes be kept sufficiently close together and that the synchronization be able to tolerate faults to a certain extent.

For fault tolerance reasons every node is equipped with its own local clock, the "physical clock," which is typically implemented by a discrete counter. The counter is incremented periodically, triggered by a crystal oscillator. As these oscillators do not resonate with a perfectly constant frequency, the clocks drift with respect to some external reference time and similarly with respect to each other. Therefore, the clocks of the nodes must periodically be resynchronized by adjusting a node's physical clock in order to keep it in agreement with the other nodes' clocks. To this end, nodes must gain information about the readings of other nodes' clocks.

By repeatedly adjusting a node's physical clock, a clock synchronization protocol implements a so-called "logical clock" for each node. The task of the synchronization algorithm is to bound the *skew*, that is, the absolute difference between the readings of the logical clocks of any two nonfaulty nodes by a small value.

Agreement: At any time, the value of the logical clocks of all nonfaulty nodes should be approximately equal.

To exclude trivial solutions to this problem one usually requires that the adjustments made to the physical clock readings are also bounded by some small value.

Accuracy: There is a small bound on the value by which a nonfaulty node's clock is changed during each synchronization interval.

The problem of clock synchronization is well understood, and quite a number of different algorithms have been introduced to provide synchronized clocks, with different requirements or assumptions about the hardware architecture, failure model, quality of the synchronized clocks, or the additional message overload.

The clock synchronization algorithm of TTP/C belongs to the so-called convergence-averaging algorithms, which are characterized by using averaging functions to calculate a new clock value from the collected readings of the other clocks. Typical averaging algorithms discard a defined number of largest and smallest clock readings to compensate for possible faulty readings and adjust the node's clock according to the mean or the midpoint of (some of) the remaining values. However, the algorithm implemented in TTP/C shows several special characteristics. First, it is completely integrated into the ordinary message exchange of the nodes. This is accomplished by exploiting common knowledge about the temporal behavior of the system: due to the TDMA-driven communication mechanism every node knows exactly at which time to expect a message by a given sending node. This knowledge, more precisely the deviation between the time a message is expected to arrive and the actual arrival time, is used to calculate estimates of the reading of the sending node's clock. The nodes, however, only keep the measured time-difference values of the four most recently received messages. At resynchronization, a node discards the largest and the smallest of these four clock readings and uses the average of the remaining two values to adjust its local clock. This procedure is known as the fault-tolerant average algorithm [14]. Moreover, the four clock readings that a node keeps for synchronization are further constrained. The arrival time is only measured if the message is considered *correct* by the receiving node. In particular, this means that sender and receiver agree on the current protocol state, which includes the current membership. As a consequence, time-difference values are only kept if the sending node belongs to the same group as the receiver. Hence, for the correct functioning of clock synchronization it is crucial that the agreement protocol of the membership service holds, that is, that all nonfaulty nodes belong to the same group. This exemplifies the deep intertwining and interesting interactions of different algorithms in TTP/C.

Apart from its peculiarities, the TTP/C algorithm is closely related to well-known synchronization algorithms, in fact it can be seen as being a variant of both the fault-tolerant midpoint algorithm of Lundelius-Welch and Lynch [15] and the interactive convergence algorithm of Lamport and Melliar-Smith [16].

15.2.4 Startup and Reintegration

The startup problem is closely related to that of clock synchronization. While synchronization has to adjust the local clocks of nodes so that they remain synchronized despite the drift of their hardware clocks, a startup algorithm is necessary to establish consistent values for the local clocks as the nodes first power up so that they quickly become synchronized. A variant of startup is restart, when synchronization needs to be reestablished after transient faults have afflicted one or more (or all) nodes.

A basic solution to the startup problem is for nodes that see no traffic for some time to send a "wake-up" message that carries their own identity. This message provides a common event that all nodes can use as a baseline for their local clocks, and the identity of the sender indicates the position in the TDMA schedule to which this time corresponds.

Of course, two nodes may decide to send wake-up messages at approximately the same time, and these messages will "collide" on the channel. In a bus-based TTA, the signals from colliding messages physically overlay on the medium, but propagation delays cause different nodes to see the signals at different times so that collision detection can be unreliable. In a star topology, the central guardians arbitrate collisions and select just one message from any that arrive at approximately the same time to forward to the other nodes. However, each central guardian arbitrates independently, so nodes can receive different messages on the two channels at approximately the same time; resolving these "logical collisions" is the task of the startup algorithm.

The startup algorithm executed in the nodes is based on two kinds of timeout parameters, the *listen timeout* τ_p^{listen} and the *coldstart timeout* $\tau_p^{\text{coldstart}}$. These timeouts are unique to each node p, and the values of each listen timeout are larger than all coldstart timeouts. Two cases are distinguished: either a starting node can (re)integrate to an already running and synchronized set of nodes, or it must initiate or wait for a coldstart to be executed. To determine whether there already is a synchronous set of nodes to integrate, a node first listens to the channels for integration messages, which are transmitted periodically during synchronous operation and that carry the current agreed protocol state, including position in the TDMA round. If the node receives such a message, it adjusts its state to the message contents and is thus synchronized to the synchronous set. If the node's listen timeout expires before an integration message is received, a coldstart is executed.

During coldstart, a node first waits to receive a special coldstart message, which is similar to ordinary messages but carries a protocol state suggested by the sending node. If no coldstart message is received before the node's coldstart timeout expires, the node sends a coldstart message by itself. If a node receives such a message, it resets its clock and waits another $\tau_p^{\text{coldstart}}$ time units for the next message (either coldstart or normal) to arrive, which will be used for synchronization. The reason why nodes synchronize only to the second (coldstart) message they receive—a mechanism called the *big bang*—is the possibility that two nodes send out simultaneous or overlapping coldstart messages. The receiving nodes will see this as a logical collision. The subsequent message, however, is deterministic, as no further collision can occur due to the unique timeouts of the nodes: the node whose coldstart timeout expires first will send first. Furthermore, as the listen timeouts are all larger than any coldstart timeout, no newly starting node may cause another collision.

In addition to collisions, the startup algorithm must deal with faulty nodes that may send "wake-up" messages at inappropriate times, masquerade as other nodes, and generally fail to follow the algorithm. The central guardians can detect and mask these faults, but increase the complexity of the algorithm since they must themselves synchronize with the nodes during startup.

Because the communication system is replicated and there are two central guardians, it is particularly crucial that a faulty node must not be able to initiate or infiltrate a startup sequence to cause the two central guardians to start at different positions in the TDMA schedule. And, of course, one of the guardians could itself be faulty. Fault-tolerant startup of a TTA system clearly constitutes a rather intricate problem as discussed in detail in Ref. [17], and hence provides a challenging subject for formal analysis.

15.3 Modeling Aspects

Formal analysis of distributed fault-tolerant algorithms is an inherently difficult task, which is due to the immense complexity of the behavior of such algorithms caused by the effects of faults and failed components. Since such analyses are generally not feasible at the implementation level of the algorithms, adequate formal models need to be developed carefully, and a major question to be solved is which details should be part of the model, and which should not. On the one hand, any formal model should reflect the real world as closely as possible. On the other hand, the amount of detail included in a formal specification greatly influences the feasibility of a mechanical proof. This is true for both model checking and approaches based on theorem proving. Regarding the first, using abstractions to limit the state space that has to be explored in a model checking analysis is a prime issue, as the run time for an experiment quickly increases as the model grows and soon becomes infeasible. Likewise, it is advantageous for theorem proving to sharpen the specification by abstracting from unnecessary or irrelevant details that would otherwise impede focusing on the main aspects of a proof.

15.3.1 Modeling Computation

Many distributed algorithms, including the central algorithms of TTP/C, proceed in a series of rounds and can easily be described as recursive functions. Correctness proofs then typically involve more or less simple forms of induction. Rushby [18] shows how such round-based descriptions can be related to models that more closely reflect the time-triggered execution of the algorithm. The approach proceeds in two steps, and involves expressing the execution of an algorithm by a set of distributed nodes at two levels of abstraction. As a first step, round-based descriptions are translated into the *untimed synchronous system model* [19], which can be done systematically. In this model, all nodes are assumed to be perfectly synchronized and operate in locked steps. The notion of time is abstractly captured by numbering the steps the nodes take. In terms of TTP/C, these steps roughly correspond to the slots defined by the protocol schedule. This is an abstraction from a clock synchronization mechanism. The second step consists of refining the synchronous model into a *timed synchronized model* and can be accomplished in a generic way independent from any given algorithm.

A standard way of specifying distributed algorithms are state-transition systems. To describe a given algorithm executed by a node in a distributed system one defines the set of states the node starts in, which messages it generates given its current internal state, and how it moves from one state to the next at the reception of a message. More formally, one needs to provide interpretations of three functions:

- $initstate(p)$, which assigns an initial state to every node p
- $msg(p, s)$, which is the message generation function and denotes the message node p sends in state s
- $trans(p, s, m)$, which is the state transition function and describes the new state node p moves to when receiving message m in state s

A snapshot of an execution of the algorithm in a distributed system at time t is then given by the set S_t of internal states of all nodes, the set M_t^s of messages generated by the nodes, and the set M_t^r of messages received by the nodes. The granularity at which one describes how these sets evolve over time characterizes the system model. In the untimed synchronous model, time is modeled at slot granularity and the execution of an algorithm can be described using the following three functions, where *Slot* represents slot numbers and *Node* denotes the type of node identifiers.

- $state^{ss}$: *Slot* × *Node* → *State*, where $state^{ss}(t,p)$ yields the internal state of node p at the beginning of slot t.
- *sent*: *Slot* × *Node* → *Message*, where $sent(t,p)$ represents the message node p sends during slot t.
- *rcvd*: *Slot* × *Node* → *Message*, where $rcvd(t,p)$ denotes the message that node p receives during slot t.

The behavior of a node in the synchronous system model is captured by relating these entities with the message generation function *msg* and the state transition function *trans*, which constitute the given distributed algorithm. However, one has to distinguish between nonfaulty and faulty behavior of a node. For nonfaulty nodes, one would assume that they start in their initial states and then synchronously perform the following steps [20]:

1. Apply the message generation function to the current state to determine the message to be sent to the other nodes. Put this message in the outgoing channels.
$$sent(t,p) = msg(p, state^{ss}(t,p))$$
2. Apply the state-transition function to the current state and the messages received through the incoming channels to obtain the new state. Remove all messages from the channels.

$$state^{ss}(t,p) = \begin{cases} initstate(p) & \text{if } t = 0 \\ trans(p, state^{ss}(t-1,p), rcvd(t-1,p)) & \text{if } t > 0 \end{cases}$$

For TDMA-based communication patterns as in TTP/C, the message generation function would yield some special value *null*, denoting no message, for all nodes that are not the scheduled sending node in slot t.

The behavior of faulty nodes and the relationship of the functions that describe the sending and the reception of messages, that is, *sent* and *rcvd*, are subject to the fault model that is adopted for the analysis of the algorithm under study. For example, if one needs to consider arbitrary, or *Byzantine*, node failures, the functions *state* and *sent* could be assumed to yield any state or message, respectively, without necessarily obeying the message-generation function or state-transition function. A *fail-silence* behavior of a node could, for instance, be modeled by defining $sent(t,p)$ to yield *null* from some slot t onwards.

As the untimed synchronous system model abstracts from the timing aspect, algorithms that specifically involve time, such as clock synchronization, are not expressible at this level. Hence, one also needs a more detailed description to model the timing behavior of nodes. In comparison to the model for the untimed synchronous level the "timed synchronized model" additionally introduces entities that formalize the local clocks of the nodes, and the state updates and generation of messages are now described in a more fine-grained way on the level of the ticks of these clocks, rather than in terms of slots.

The number of functions used to describe a time-triggered system is doubled compared to the synchronous system model. In addition to the function *state* that describes the internal state of a node, a function *message* is introduced that denotes the message that is available to a node at a particular moment. This function extends the notion of message reception known from the synchronous system model. Still, the function *rcvd* denotes the message that arrives at a node during a slot; however, in the time-triggered model one has to take into account whether the message arrives *in time*. If it does, *message* will correspond to *rcvd*, otherwise it will yield the empty message. Moreover, functions *send_time* and *arr_time* are introduced to denote the (realtime) instant at which a node sends and receives a message, respectively.

More formally, a time-triggered system executing a distributed algorithm is described through the following functions:

- $state^{tt}$: *ClockTime* × *Node* → *State*, where $state^{tt}(T, p)$ yields the internal state of node p when its local clock reads the clock time value T.
- *message*: *ClockTime* × *Node* → *Message*, where $message(T, p)$ yields the message that is available to p when its logical clock reads the clock time value T.
- *sent*: *Slot* × *Node* → *Message*, where $sent(sl, p)$ represents the message node p sends during slot *sl*.
- *rcvd*: *Slot* × *Node* → *Message*, where $rcvd(sl, p)$ denotes the message that node p receives during slot *sl*.
- *send_time*: *Slot* × *Node* → *realtime*, where $send\_time(sl, p)$ denotes the realtime instant at which the sender p of slot *sl* sends its message.
- *arr_time*: *Slot* × *Node* → *realtime*, where $arr\_time(sl, p)$ denotes the arrival time of the message p receives in slot *sl*.

The entities above involve two notions of time that have to be distinguished: *realtime* and *clocktime* [16]. Realtime is an abstract notion and not directly observable in the system, while clocktime is the local notion of time available to a node by way of its clock. The (logical) clock LC_p of a node p, maps real-time instants t to clocktime values T, such that $LC_p(t)$ denotes the reading of p's clock at realtime t.

As in the synchronous system model, the entities above need to be related to those that actually model the distributed algorithm under study. In contrast to synchronous world, one does not only have to formalize how a node proceeds, but also when it is to take its steps. To this end, a function $schedule(sl)$ is introduced that yields for every slot *sl* the clock time instant at which a node is to start that slot. Slots can conceptually be divided into a "communication phase," where nodes send and receive messages, and a "computational phase," in which the nodes update their internal states [18]. These concepts are reflected using a function $cmp\_start(sl)$, which denotes an offset into slot sl at which a node ends its communication phase and begins its computation phase.

Using these functions, the internal state of a nonfaulty node can be modeled as follows:

- At the start time of the first slot, p is in its initial state:

$$state^{tt}(schedule(0), p) = initstate(p)$$

- During the communication phase of a slot, the state of a node remains unchanged:

$$\begin{aligned}&state^{tt}(T, p) = state^{tt}(schedule(sl), p)\\&\text{if } schedule(sl) \leq T \leq schedule(sl) + cmp\_start(sl)\end{aligned}$$

- At some point during its computation phase, p updates its internal state according to its state-transition function, which is applied to the state of p at the end of its communication phase, and the message that is available to p at that time:

$$\begin{aligned}&state^{tt}(schedule(sl+1), p) = trans(p, state^{tt}(T, p), m)\\&\quad\text{where } T = schedule(sl) + cmp\_start(sl)\\&\quad\text{and } m = message(T, p)\end{aligned}$$

To characterize the value of $message(T, p)$ one needs to take into account whether node p received a message during its latest communication phase; if so, it corresponds to $rcvd(sl, p)$, otherwise the message is *null.*

$$\begin{aligned}&message(T, p) = rcvd(sl, p)\\&\text{if } schedule(sl) \leq arr\_time(sl, p) < schedule(sl) + cmp\_start(sl)\end{aligned}$$

15.3.2 Modeling Time

The TTA startup algorithm has been subject to several formal analyses, all of which are based on model-checking techniques. A common aspect of model checking analyses is that a large part of the effort is devoted to keeping the size of the model in a range that is computationally feasible, while at the same time aiming for realistic formal models that do not employ oversimplifying abstractions.

For the startup problem, a central aspect is how to adequately capture the notion of time in the formal model. Timed automata are a successful formalism for the verification of real-time systems and treat time in the most realistic formal way as a continuous variable. Still, model checking timed automata are decidable, and there are specialized model checking tools such as Kronos and UPPAAL. Lönn [21] considers startup algorithms for TDMA systems similar to TTA and verifies one of them using UPPAAL.

As model checking timed automata are computationally complex, other formalisms and abstraction might be needed when emphasis is shifted from timing aspects to analysing elaborate fault behaviors or a large number of different fault scenarios. In their analysis of TTA startup, Steiner et al. [9] use an abstraction employing discrete time that treats slots as indivisible units. Similar to the synchronous model described in the previous section, the model abstracts from the concrete duration of the slots and from how much the slots at different nodes are offset. The nodes measure time by counting off slots in the TDMA schedule and the collective behavior of a cluster of nodes is modeled as the synchronous composition of discrete systems.

Dutertre and Sorea describe an approach that preserves time as a continuous variable, but makes timed systems amenable to analysis also by model checkers for discrete transition systems [7]. However, as the state space becomes infinite, satisfiability modulo theories-based (SMT-based) bounded model checkers are necessary. They apply their approach to verifying the TTA startup algorithm using k-induction.

Central to their approach is the concept of *event calendars*, which is well known from the area of discrete-event simulation. An event calendar stores the times at which certain events, such as the reception of a message by some node, will occur in the future. Two types of transitions of such *calendar automata* are distinguished: time progress transitions, which advance time, and discrete transitions, which update state variables. At every step, only one type of transition is enabled: either time is advanced to the instant when the next event is scheduled to occur—as controlled by the event calendar—or one of the discrete transitions of those events that occur at the current time is taken. Additional constraints ensure that time is always maximally advanced and that there are no infinite sequences of discrete transitions without time progress. The advantage of this approach is that all variables, including time, evolve in *discrete* steps, and thus there is no need to approximate continuous dynamics by allowing arbitrarily small time steps. In effect, this reduces the number of possible transitions and thus the size of the system model.

Using calendar automata, the models of components of a system, such as the nodes in TTA, are asynchronously composed, and synchronous communication is modeled by the sequential application of discrete transitions. Pike observes [22] that through using a synchronous composition, multiple transitions may be applied simultaneously. Building upon Dutertre and Sorea, he introduces the synchronizing timeout automata model. In this model, subsets of state variables are distinguished, which represent portions of the state of the system that are updated synchronously. As in calendar automata models, additional rules ensure progress of the system. Furthermore, the clock in a synchronizing timeout automata model can be conservatively removed, as the amount by which it is updated, and thus its current value can be obtained from other components of the model. This optimization reduces both the state space and the number of transitions of the system, and thus larger models can be analyzed. Pike applies his approach to a reintegration protocol implemented in a related distributed fault-tolerant architecture called Spider [23].

15.3.3 Modeling Faults

Modeling fault-tolerant systems includes careful consideration of the *fault hypothesis* relative to which the systems provide their fault tolerance properties. The fault hypothesis comprises the assumptions about the kinds of faults that can occur, the frequency of such occurrences, and the total number of faults that can be tolerated.

Hybrid fault models [24,25] distinguish various kinds of faults with different levels of severity. For example, faults that can consistently be detected by nonfaulty nodes are less severe, and are called *benign*, or *manifest* faults. *Symmetric* faults can generally not be detected immediately, but every nonfaulty node observes the same behavior. The most severe fault type are asymmetric faults, which may exhibit different behavior to different nonfaulty nodes.

The primary fault hypothesis for TTA is the *single fault hypothesis*, that is, the architecture can tolerate arbitrary failures of one of its components. Recently a new, broader fault hypothesis that addresses transient failures of multiple components has been discussed [26]. It relies on a stronger clique resolving algorithm that allows the system to reconfigure after a transient upset by excluding faulty nodes.

There are further design choices with respect to the components to which faults are attributed. For example, fault models can include *link* faults to describe failures of the communication channels. Alternatively, channels can be seen as belonging to either the sender or the receiver, and channel faults are abstracted by attributing them to particular nodes. The analysis of TTA group membership algorithm, for instance, only considers faulty nodes, which can be either send-faulty or receive-faulty. Pike et al. [27] abstract node faults as being ones only affecting a node's ability to send, such that a failure of a node to receive a message eventually manifests itself as a send fault of that node.

Deductive analyses based on theorem proving can be carried out relative to rather general fault models. Pike et al. [27] propose various kinds of abstractions to facilitate modeling and analyzing fault-tolerant systems, including abstractions to describe faults and their effects on the correctness of the individual messages sent and received by nodes. In their model, there are two types of abstract messages: *accepted* messages carry an associated value *m*, the message data, which can be extracted by the receiving node, while all detectably incorrect messages are abstracted as a single *benign* message. Faults are abstractly modeled by a function *send*, which describes the abstract message that nodes *r* receive from a sender *s*, depending on its fault status:

$$\mathit{send}(m, \mathit{status}, s, r) = \begin{cases} \mathit{accepted}(m) & \text{if } \mathit{status}(s) = \mathit{good} \\ \mathit{benign} & \text{if } \mathit{status}(s) = \mathit{benign\ faulty} \\ \mathit{sym}(m, s) & \text{if } \mathit{status}(s) = \mathit{symmetric\ faulty} \\ \mathit{asym}(m, s, r) & \text{if } \mathit{status}(s) = \mathit{asymmetric\ faulty} \end{cases}$$

The functions *sym* and *asym* are uninterpreted in the sense that no further properties are assumed as to which particular messages are received. However, in case of sym, it can be deduced that all nodes receive the same, albeit unknown, message, while for two receivers r and r' the message denoted by $\mathit{asym}(m, s, r)$ is generally not equal to $\mathit{asym}(m, s, r')$. Thus, the fault behavior of asymmetrically faulty nodes can be left completely unspecified.

In contrast to this declarative approach to fault modeling, the effects of faults must be modeled explicitly in a model checking context. For example, to model asymmetric send faults, one needs to allow for a node to send different messages to different nodes. Specification languages for model checkers usually support to express the nondeterministic choice of an element from a given base set, and asymmetric send-faulty nodes can thus be modeled by nondeterministically choosing a message for each receiving node. Note, however, that can significantly increase the state space of the model, as the model checker has to analyze every possible combination of messages.

15.4 Verification Techniques

Fault tolerance properties of the TTA and its underlying communication protocol TTP/C have been formally analyzed with various techniques. This section highlights approaches taken in both theorem proving and model checking analyses.

15.4.1 Theorem Proving

In addition to carefully developed models, an appropriate organization of proofs is indispensable to cope with the complexity of deductive verification of fault-tolerant systems. The techniques described in the sequel include approaches that aim to decompose proofs into manageable steps and application of specialized proof techniques.

15.4.1.1 Refinement

In order to facilitate the deduction, the formal proofs are generally decomposed into a series of smaller steps. An instance of this approach is refinement-style deduction, where one starts by specifying the desired property in an abstract form; a correctness proof for that property is then based on a number of assumptions about entities of the abstract model. Subsequently, more detail is added to this initial abstract model, for instance by providing concrete interpretations for certain abstract entities. A proof that the desired property also holds for the refined model can then be inherited from the abstract model by demonstrating that the concrete interpretations satisfy the assumptions on which the abstract proof relies.

The analysis of the fault tolerance properties of the central guardians in a star-based TTA system follows this general scheme [28]. In order to prove that the central guardians extend the class of faults that can be tolerated by the system, a series of formal models is developed, which are organized in a hierarchical fashion.

Each of the models contributes a small step toward proving the desired correctness properties. The steps themselves are each based on a set of assumptions, or preconditions, and in each model layer i one establishes a theorem of the form

$$assumptions_i \Rightarrow properties_i$$

The idea is to design the different models in such a way that the properties on one level establish the assumptions on the next. Ultimately, the models are integrated and the reasoning is combined, yielding a chain of implications of roughly the following kind:

$$\begin{aligned} &assumptions_0 \Rightarrow properties_0 \Rightarrow assumptions_1 \Rightarrow properties_1 \\ &\Rightarrow \cdots \Rightarrow properties_f \end{aligned}$$

The final properties, $\mathit{properties}_f$, correspond to the desired main correctness properties of the TTP/C communication, while the initial assumptions, $\mathit{assumptions}_0$, describe what constitutes the basic fault hypothesis.

The most abstract model describes the reception of messages by the nodes. Here, the various actions that nodes take in order to judge the correctness of the received message are formalized. This amounts to considering the transmission time and the signal encoding of the message, and the outcomes of several consistency checks. The main correctness properties of communication are then expressed in terms of these notions. The assumptions of this model layer concern requirements about the functionality of the communication channels. In particular, they describe properties of the messages that a channel transmits, such as signal encoding or delivery times, and reflect the hypothesis about possible faults of the communication network. In essence, this model establishes a proposition that informally reads as follows:

$$\mathit{general\_channel\_properties} \Rightarrow \mathit{Validity} \wedge \mathit{Agreement} \wedge \mathit{Authenticity}$$

The next level models the transmission of messages through channels that are not equipped with guardians. The goal is then to derive the assumptions of the basic model, as covered by the expression *general_channel_properties*. However, in order to do so, a strong hypothesis on the types of possible faults of nodes is necessary. This strong fault hypothesis requires, for instance, that even a faulty node does not send data outside its sending slot, and nodes never send correct messages when they are not scheduled to do so.

$$\mathit{strong\_fault\_hypothesis} \Rightarrow \mathit{general\_channel\_properties}$$

Guardians are employed to transform arbitrary node faults into faults that are covered by the strong fault model. Thus, the strong fault hypothesis can be replaced with weaker assumptions about the correct behavior of the guardians. The functionality and the properties of the guardians are formally specified in the third model of the hierarchy, where the following fact is established:

$$\mathit{weaker\_fault\_hyp.} \wedge \mathit{generic\_guardian} \Rightarrow \mathit{general\_channel\_properties}$$

The model of the guardians is generic, as it does not, for instance, stipulate the type of guardian to be used in the communication network. The final level of the hierarchy models each of the two typical topologies of a TTP/C network: the bus topology and the star topology. In the former, each node of the network is equipped with its own local bus guardian, one for each channel, while in the latter the guardians are placed into the central star-coupling device of the channels. In this model layer it is shown that the properties of the guardians are independent from the choice of a particular topology, given that both the local bus guardians and the central guardians implement the same algorithms. Hence, the following facts are established:

$$\begin{array}{rcl} \mathit{local\_bus\_guardian} & \Rightarrow & \mathit{generic\_guardian} \\ \mathit{central\_star\_guardian} & \Rightarrow & \mathit{generic\_guardian} \end{array}$$

15.4.1.2 Generic Verification

A verification strategy similar to refinement is that of "generic verification." As in the refinement approach, an abstract formulation of the problem is developed and a proof of correctness is based on certain abstract assumptions. The difference is, however, that this abstract model is intended to cover a whole class of similar problem instances. In this sense, the verification is generic, as it is valid for various implementations, provided that these satisfy the abstract assumptions.

The clock synchronization algorithm of TTP/C has been analyzed using this approach [29] and follows similar developments for other algorithms in the long history of verification for clock synchronization. It was observed by Schneider [30] that the correctness arguments of averaging algorithms are quite similar. This class of algorithms can be captured in an abstract way by introducing the concept of a *convergence function Cfn* to describe the way the adjustments to a node's physical clock are computed. To carry out the re-synchronization, a node p has to obtain estimates of the readings of the other nodes' clocks in one way or the other, and the values are stored in an array Θ_p. The value $\Theta_p(q)$ then represents p's estimate of q's clock reading at the time of re-synchronization. The convergence function is applied to this array of clock readings, such that $Cfn(p, \Theta_p)$ is the new, corrected reading for p's clock. The difference between this value and the current reading of p's clock yields the amount by which p has to adjust its clock.

Schneider stated several rather general assumptions on the convergence function and showed that they are sufficient to prove the correctness of several averaging algorithms. Subsequently, Shankar used the EHDM system to mechanically verify Schneider's proof [31], and Miner [32] has further improved the constraints and the organization of the proof itself.

The agreement property for clock synchronization states that at all times the difference of the clock readings of any two nonfaulty nodes p and q is bounded by a fixed value Δ:

$$|LC_p(t) - LC_q(t)| \leq \Delta$$

The proof of the agreement property is generally accomplished through mathematical induction on the number of resynchronization intervals. The induction hypothesis states that at the beginning of each interval, the skew between any two clocks is bounded by some value $\Delta_0 < \Delta$. Then it is shown that during the next interval, during which the clock readings may drift apart from each other, the skew does not exceed Δ. Finally one has to prove that the application of the convergence function brings the clocks closer together again, so that the next interval can start with the clocks being within Δ_0 of one another. The latter step is the harder one; the former simply imposes certain constraints on the maximum precision that can be achieved given concrete values for the drift rate of the clocks and the length of a synchronization interval.

Schneider's assumptions essentially express in a generic way the properties that are necessary to accomplish the inductive step in the proof of *agreement*. Some of them concern the interrelationships among the various quantities involved in the synchronization algorithm, such as the assumed bounds on the drift rate of the clocks, or the maximum error made when estimating a remote node's clock. The more important of the assumptions are concerned with the behavior of the convergence function that a clock synchronization algorithm exploits. The usefulness of these conditions is for the most part due to their isolation of purely mathematical properties from other concepts such as, for example, faulty components.

One of the central assumptions is called *precision enhancement* and is used to bound the skew between the two applications of the convergence function. The actual bound depends on the skews between the values in the array of estimated clock readings. Given two such arrays θ and γ used by two nodes p and q, respectively, precision enhancement states that the absolute values of the convergence function applied by p and q do not differ by more than a quantity $\Pi(X, Y)$, provided that corresponding entries in θ and γ differ by no more than X and the values in θ and γ, respectively, fall within a range Y. Furthermore, it is required that $\Pi(X, Y) < Y$ for the precision to be truly enhanced.

> *Precision Enhancement*: Given a subset C of the n nodes such that $|C| \geq n - f$, where f is the number of faults to be tolerated, there is a bound $\Pi(X, Y)$ such that
>
> $$\begin{array}{ll} \text{if} & \forall l \in C : |\gamma(l) - \theta(l)| \leq X \text{ and} \\ & \forall l, m \in C : |\gamma(l) - \gamma(m)| \leq Y \text{ and } |\theta(l) - \theta(m)| \leq Y \\ \text{then} & |Cfn(p, \theta) - Cfn(q, \gamma)| \leq \Pi(X, Y) \end{array}$$

The *Precision Enhancement* condition involves a subset C of the n nodes from which readings are stored in the arrays θ and γ. The elements of C must satisfy the preconditions of *Precision Enhancement*, and it is required that C contains at least $n - f$ elements. For the algorithm to tolerate f arbitrary (Byzantine) faults it is crucial that n is at least $3f + 1$ (cf. Ref. [33]). This ensures that the sets of readings used in the convergence function by two nodes overlap.

The intuitive interpretation of C is the set of readings from non-faulty clocks. However, the properties as stated do not directly enforce this interpretation of C; in fact, no distinction is made between faulty and nonfaulty clocks. In this way Schneider's model is also applicable to synchronization protocols that may use readings from faulty clocks, or possibly disregard readings from nonfaulty ones—as the TTP/C algorithm does—but ensure that the readings actually used do satisfy certain mathematical constraints.

The application of Schneider's general results for the clock synchronization algorithm of TTP/C, proceeds in three steps. First, a ground model of the algorithm is developed that follows closely the definitions in the informal protocol specification [34]. In an intermediate step this ground model is transformed into an equivalent, but more abstract version that describes the clock synchronization algorithm in terms of Schneider's generic concepts. Finally, the synchronization algorithm as stated in the abstract model is proved to indeed satisfy the various conditions of Schneider's generic proof.

One of the crucial steps in the formal analysis of the TTP/C algorithm is defining an appropriate interpretation of the set C mentioned in the definition of *Precision Enhancement*. Traditionally, the set C is interpreted as the set of nonfaulty nodes. This interpretation is feasible under a fault hypothesis that at all times at most f nodes are faulty. The rationale behind this definition is the implicit assumption that nodes can obtain the clock readings of a non-faulty nodes with only small errors, and that two nodes obtain approximately the same results when reading the clock from the same nonfaulty node. The important aspect about the clock readings, which the preconditions in the definition of Precision Enhancement emphasize, is their quality rather than their origin. As a node in the TTP/C algorithm only keeps the readings of the last four nodes from which it has received a message, n cannot be taken to be the number of nodes of the system and let C denote the set of nonfaulty nodes. This is because one would not be able to establish suitable bounds X and Y such that the preconditions of Precision Enhancement are satisfied also for those nonfaulty nodes from which there are no clock readings stored in a node's queue.

For the TTP/C instance, C is therefore defined as the intersection of the sets of those nodes from which clock readings are available in any node's queue. To establish the proof of Precision Enhancement for TTP/C it must then be demonstrated that C contains at least three elements, that is, any two nonfaulty nodes p and q must have clock readings from at least three common nodes in their respective arrays [29].

15.4.1.3 Disjunctive Invariants

The agreement property of group membership, that is, that the algorithm maintains a consistent view among the nodes about which other nodes are operating correctly, is an *invariant*, also called a *safety property*, which holds for all reachable states of the system. Traditionally, such invariant properties are verified by some form of induction proof: one demonstrates that the property holds in the initial state(s) and that all state transitions preserve the property. However, the desired properties are rarely inductive and hence, in order to establish the proof of the induction step, have to undergo a process of strengthening by conjoining additional properties. In turn, these additional properties, too, have to be invariants. Usually, this process has to be repeated several times before the induction proof can be accomplished.

Rushby proposed a method of proving invariant properties based on a symbolic forward reachability analysis of the possible states of the algorithm [35]. This method facilitates the construction of inductive invariant properties by using *disjunctive* invariants, and it was applied in the verification of the group membership algorithm of TTP/C [6].

Starting from an initial state, the state updates carried out by the nodes when executing one step of the algorithm are examined repeatedly. At every step, the states of all nodes are described in a uniform way to form a certain *configuration* of the algorithm. The execution of one step of the algorithm then corresponds to a transition from one configuration to another. The set of configurations and transitions can naturally be illustrated through a diagrammatical representation, the *configuration diagram*.

The configurations of the system form the nodes of this diagram, and arrows denote transitions from one configuration to another and are labelled with so-called transition conditions. The diagram can be seen as a graphical representation of a large part of the proof of the correctness properties, since each transition corresponds to a lemma, which embodies the assertion that one step in the execution of the algorithm leads the system to move from one configuration to another. Configurations are parameterized by the time *t* and describe the global state the system is in. Configurations can have additional parameters, such as nodes whose internal states are different from those of other nodes in the system, or additional entities necessary to describe the system state. The labels of transitions express the preconditions for the system to move from one configuration to another. The transition conditions leading from one configuration need not necessarily be disjoint, but one has to show that they are complete in the sense that their disjunction is true.

A diagram for an algorithm such as TTP/C group membership can be developed systematically by repeatedly applying the following steps:

1. Start with defining some initial configuration that typically contains all the initial states.
2. Choose one of the configurations and invent some transition conditions for it. To this end, one must analyze the algorithm to deduce the possible branches the algorithm can take. In order to ensure coverage of all cases one must prove that the disjunction of all transition conditions is true.
3. Then, for each new transition condition, symbolically simulate one step of the algorithm in the given configuration.
4. Now decide whether the result of the simulation becomes a new configuration or whether it is a variant or generalization of an already existing one. In either case, the validity of a transition, that is, that the algorithm indeed takes a step from one configuration to another, must be proved.
5. Steps 2–4 must be repeated for each configuration and each transition condition until the diagram is closed.

The definitions of the configurations must be chosen such that the correctness properties of the algorithm can be proved. This requires that the description of each configuration implies the desired safety properties and that the disjunction of the transition conditions leading from any one configuration evaluates to true; this ensures that there is no other configuration the system can possibly get into.

There are several benefits to this approach: the configuration diagram can serve as a comprehensible explanation of the algorithm, provides further insight in its functioning, and allows for straightforward what-if analyses. In such analyses, the consequences or effects of small changes, for instance to the algorithm itself or to the fault model, are examined. The graphical representation of the structure of the membership algorithm lets one trace the impact of the change through the diagram in order to identify the situations where the algorithm might fail. Last, but not least, the analysis does not depend on the size of the system, that is, the number of nodes involved. In fact, since the problem size is an uninterpreted parameter of the model, that is, represents a fixed but arbitrary value, the analysis is valid for any number of nodes.

15.4.1.4 Assume–Guarantee Reasoning

As a consequence of the tight integration of services in the TTP/C protocol suite, clock synchronization and group membership both rely on the correct operation of the other. Obviously, for group membership to work the nodes must be synchronized in order to be able to send and receive messages. Likewise, clock synchronization also depends on group membership, because when collecting estimates of remote clocks, the nodes use only messages from senders that belong to the same group. Therefore, the question arises whether any correctness statement is meaningful if the formal analysis of a particular service is carried out in isolation and is based on assumptions about the correctness of others, as this kind of circular reasoning usually is unsound.

There is an escape for TTP/C, however, because the circular dependency can be broken by dividing the reasoning according to the synchronization intervals of the protocol execution. In fact, for group membership to work properly in the ith interval, the ith resynchronization must have been correct, which in turn relies on group membership having worked correctly in interval $i - 1$, and so forth. Rushby [36] conjectures that a proof rule introduced by McMillan [37] can be applied to accomplish an integrated proof of correctness for group membership and clock synchronization in an assume–guarantee style. McMillan's rule states that if a component X_1 guarantees that a property P_1 is true at time t provided that another property P_2 is always true upto time $t - 1$, and conversely a component X_2 guarantees that P_2 is true at time t provided that P_1 is always true upto time $t-1$, then the composition of X_1 and X_2 guarantees that the conjunction of P_1 and P_2 is always true. The rule furthermore allows both the premises and the conclusion to be relative to the validity of some "helper property" H. It is easy to see how this rule could, in principle, be instantiated to the membership and clock synchronization problem.

The approach to an integrated analysis described in Ref. [29] indeed resembles this style of reasoning, although a formal correspondence has not been shown. The goal is to prove, by way of induction, that in all synchronization intervals i both clock synchronization and group membership work correctly, denoted $cs(i) \wedge mem(i)$. The verification of group membership described in Ref. [6] provides a proof for the second conjunct. The first, however, is a bit more involved, as the induction step for clock synchronization requires that the nodes base their calculation of clock adjustments on a sufficiently large set of common messages, which relies on the availability of the group membership service. This requirement can be captured by some predicate $cs\_req(i)$, such that the induction step for clock synchronization actually reads

$$cs\_req(i) \wedge cs(i) \Rightarrow cs(i+1)$$

The property $cs\_req(i)$ must therefore be proved from the correctness of membership in the ith interval, $mem(i)$. However, these facts are expressed on different levels of abstraction: the latter is proved in the untimed synchronous system model (in order to abstract from clock synchronization), while the former uses the timed synchronized model. Rushby [18] showed that the untimed synchronous model is a sound abstraction of the timed model, but the proof relies on the clocks of the nodes being synchronized. Hence, in order to prove $cs\_req(i)$ from the correctness of membership, one additionally needs the clock synchronization property in the ith interval:

$$mem(i) \wedge cs(i) \Rightarrow cs\_req(i)$$

Provided that the clocks are initially synchronized, the overall induction proof can be accomplished using this additional lemma [29].

15.4.2 Model Checking

Model checking is an attractive technique as the verification process is essentially automatic. However, methods based on exhaustive state exploration require that the state space of the system being analyzed is finite. Bounded model checkers encode the system model into logical formulae and search for counterexamples of a given property using a satisfiability solver. Tools such as BarceLogic [38], CVC Lite [39], MathSAT [40], Yices [41], or Z3 [42] integrate satisfiability solving with decision procedures for a combination of theories, including, for instance, linear arithmetic over reals and integers, and thus provide *satisfiability modulo theories*. These *SMT-solvers* can be used to handle infinite systems in bounded model checking. Bounded model checking is a technique primarily used for refutation rather than verification. It can be extended, however, to prove *safety* properties by a method known as *k-induction*.

15.4.2.1 State Space Exploration

As model checking is an automatic technique that can provide counterexamples to a property for failed verification attempts, it can be used beneficially in the design of fault-tolerant algorithms. During the design phase variations of an algorithm or its parameters are explored against several execution scenarios or different fault cases. The challenge is to get quick feedback of model checking experiments on a usefully large number of such scenarios to allow an interactive exploration of the design space. To this end, attention has to be paid to keeping the state space of the models at a feasible size. As a design becomes consolidated, attention shifts from exploration to verification and the challenge for model checking becomes one of covering a truly exhaustive set of scenarios for a realistically accurate model in reasonable time.

Steiner et al. [9] describe an approach to control the size of the state space in an analysis of a new startup algorithm for TTA, so that a single model can be used both for exploring various alternatives of the algorithm and for verifying the robustness of the final design. Faults vastly increase the state space that must be explored in model checking. Faults introduce genuinely different behaviors, but can also produce states that differ in irrelevant ways in the sense that they are distinguished by the model checker, but do not have different effects on the system behavior. For example, the concrete state of a faulty node might be irrelevant once the correct components have excluded this node from further consideration. A valuable trick in modeling fault-tolerant algorithms is to set the states of faulty components to fixed values once they can no longer affect the behavior of the system. In the analysis of the startup algorithm, this mechanism proved essential to reduce the state space for very large models, although it had only little effect on small or medium-sized models.

To control the various ways a faulty node can influence the system, the model is parameterized by a special variable that selects the fault modes that a faulty node may exhibit. A faulty node is simulated as one that can send arbitrary messages in each slot and the possible outputs of such a faulty node are classified into six different fault degrees. For example, a fault degree of 1 allows a faulty node only to fail silently, while the highest fault degree of 6 allows a node to send an arbitrary combination of coldstart messages and normal messages with correct or incorrect semantics, noise, or nothing on each of the two channels. Using smaller values for the fault degree and the number of nodes considered allows trading execution time required against thoroughness of the exploration performed by the model checker.

However, for verification one is interested in "exhaustive" simulation of faults. Exhaustive fault simulation means that all hypothesized fault modes are modeled and all their possible scenarios are examined. For startup, this means model checking the algorithm with the fault degree set to 6 for a reasonable-sized cluster. Steiner et al. were successful verifying the correctness of TTA startup, viz., that within a bounded time all correct nodes will become active and have a consistent view on the current position in the schedule, for upto five nodes [9].

15.4.2.2 Infinite-State Bounded Model Checking

Bounded model checking [43] is basically a technique to check whether a state-transition system contains an execution that reaches a state in k steps that violates a given property P. This problem can be translated into one determining the satisfiability of the formula $F = I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$, where I is a predicate defining the initial states of the system, and T is the transition relation. For finite systems, I and T are encoded as Boolean formulae and satisfiability solvers are used to check the formula. Infinite-state bounded model checkers rely on decision procedures that solve quantifier-free first-order formulae over a combination of decidable theories. As these SMT-solvers have become ever more efficient, bounded model checking has gained increasing attention as a method for analyzing systems.

If the property to check is a safety property of the form $\square P$—meaning P should be true in all reachable states of the system—and a sequence of states satisfying the formula F exists, then one can conclude that $\square P$ is not true. However, the converse does not hold, that is, if F is not satisfiable, one cannot conclude that $\square P$ is true, because there might be no execution traces of length k at all, or the property might be true for x_k but violated in some earlier state. One can account for these cases by iteratively increasing the depth k and checking for every step whether P is violated in any of the states $s_0, \ldots, s_k$; however, neither method can be used to *prove* $\square P$, since the failure to find counterexamples for some length does not preclude that there some longer ones. Note that bounded model checking is not limited to safety properties; this example is used to facilitate the illustration.

To actually prove a safety property P bounded model checking is extended to *k-induction*. For a fixed depth k, one needs to show in the base case that all states of a sequence of length k starting from an initial state satisfy P. The induction step consists of proving for any sequence of states of length $k + 1$, that if the first k states satisfy P, then so does the last one. In most cases, the invariant P needs to be strengthened to make it inductive. An easy way to do so is to increase the depth parameter k. However, the effort to solve the corresponding formulae increases exponentially with k, and the problem will eventually become infeasible. Alternatively, one can strengthen the invariant with an additional property Q and prove both the base case and the induction step under the additional assumption that all states of the sequence of states considered also satisfy Q. To be sound, Q itself must be a safety property, and thus plays the role of a lemma in the proof of P.

Dutertre and Sorea have used the SAL system [44] to apply infinite-state bounded model checking with k-induction to the TTA startup algorithm [7]. Their study demonstrated that sophisticated effort is necessary to cope with the complexity of the proof. For example, even for a simplified model of the algorithm with only two nodes that are also assumed to be reliable, which allows to use guardians of only very limited functionality, the proof for the safety property fails for values of k upto 20. Using three simple lemmas, the proof succeeds for two nodes, but already becomes computationally infeasible when the number of nodes is increased to three. To accomplish the proof for higher numbers of nodes, strengthen the property using an abstraction of the algorithm. To obtain an appropriate abstraction, they apply the technique of *disjunctive invariants*, which has also been used in the deductive verification of the TTA group membership algorithm. By examining how the startup algorithm works, a number of abstract configurations and corresponding abstract transitions are defined. To prove the abstraction correct, a monitor module is built, which checks that there is a transition in the abstract model for every transition that the concrete algorithm can take. The abstraction approach turned out to be feasible to prove the desired safety property for a fault-tolerant version of startup with one reliable guardian and at most one faulty node for systems of upto 10 nodes.

15.5 Perspectives

The safety-critical nature of the range of applications envisaged for the TTA demands a high level of confidence in the correctness of the underlying principles. Several of TTA's key aspects have therefore been subjected to formal analyses in order to contribute to achieving the required degree of reliability. Fault-tolerant algorithms are, however, inherently difficult to analyze because of the possibly unrestricted behavior of faulty components, which immensely increases the complexity of the analyses. Hence, formal verification has focused on the most crucial algorithms of the TTA that warrant the effort. To reduce complexity, some aspects have been analyzed only under restricted fault models, like the group membership algorithm, on a higher level of abstraction, or in a simplified form, such as the startup algorithm. The challenge remains to provide comprehensive analyses of TTA's fault tolerance properties, including the interactions and interdependencies of the individual algorithms.

In addition to formally verifying the central protocol services, future research will also have to address the higher-level properties of TTA, such as, for instance, partitioning, that are not provided by a particular algorithm, but rather emerge from the interplay of the various protocol services and the architectural properties of the TTA. To this end, suitable formal models are necessary that satisfactorily capture these emergent properties, and adequate compositional verification techniques need to be developed to enable formal proofs of the correctness claims.

Furthermore, future formal analyses will need to widen the scope from the verification of key components of the TTA on rather abstract levels of formal models to providing a complete chain of correctness arguments from the level of applications built on top of a TTA to the silicon implementation of its protocol services. There are efforts in these directions for systems similar to TTA. For example, the SPIDER architecture is being developed at NASA Langley by a team of both electronics engineers and formal methods researchers who apply theorem proving in a rigorous design verification of the underlying communication system to ensure a tight connection between the verified formal models and the hardware implementation [23,45]. In the context of FlexRay, the VeriSoft project aims at a pervasive verification of the full range of an automotive system, including models at the application layer, the communication layer and the device drivers for the bus interfaces of a real-time operating system, and the gate-level implementation of FlexRay bus interfaces [46].

Ultimately, a major research challenge consists in extending the formal modeling and verification techniques toward a state where formal analyses can eventually be used as a basis in the certification of time-triggered systems.

References

1. H. Kopetz. The time-triggered approach to real-time system design. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, Editors, *Predictably Dependable Computing Systems*. Springer-Verlag, New York, 1995.
2. H. Kopetz. The time-triggered architecture. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, Kyoto, Japan, April 1998, pp. 22–31.
3. H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.
4. H. Pfeifer, D. Schwier, and F. von Henke. Formal verification for time-triggered clock synchronization. In C. Weinstock and J. Rushby, Editors, *Dependable Computing for Critical Applications (DCCA-7)*, San Jose, CA, *Dependable Computing and Fault-Tolerant Systems*, 12:207–226, January 1999.
5. A. Bouajjani and A. Merceron. Parametric verification of a group membership algorithm. In W. Damm and E.-R. Olderog, editors, *Proceedings of the Seventh International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Oldenburg, Germany, Lecture Notes in Computer Science, 2469:311–330, 2002.
6. H. Pfeifer. Formal verification of the TTP group membership algorithm. In T. Bolognesi and D. Latella, editors, *Formal Methods for Distributed System Development—Proceedings of FORTE XIII/PSTV XX*, Pisa, Italy, October 2000, pp. 3–18.
7. B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In Y. Lakhnech and S. Yovine, Editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Grenoble, France, Lecture Notes in Computer Science, 3253:199–214, 2004.

8. A. Merceron, M. Müllerburg, and G. Pinna. Verifying a time-triggered protocol in a multi-language environment. In W. Ehrenberger, Editor, *Proceedings of the 17th International Conference on Computer Safety, Security and Reliability (SAFECOMP)*, Heidelberg, Germany, Lecture Notes in Computer Science, 1516:185–195, 1998.
9. W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, June 2004.
10. H. Pfeifer and F. von Henke. Modular formal analysis of the central guardian in the time-triggered architecture. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *Proceedings of the 23rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Potsdam, Germany, Lecture Notes in Computer Science, 3219:240–253, September 2004.
11. H. Kopetz. The time-triggered (TT) model of computation. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998, pp. 168–177.
12. G. Bauer, H. Kopetz, and W. Steiner. Byzantine fault containment in TTP/C. In *Proceedings of the First International Workshop on Real-Time LANs in the Internet Age (RTLIA)*, Vienna, Austria, June 2002, pp. 13–16.
13. G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in the time-triggered architecture. In *Proceedings Sixth International Symposium on Autonomous Decentralized Systems (ISADS)*, April 2003, Pisa, Italy, pp. 37–44.
14. H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, 1987.
15. J. Lundelius-Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.
16. L. Lamport and P. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.
17. W. Steiner. Startup and recovery of fault-tolerant time-triggered communication. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, 2004.
18. J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.
19. F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, 1991.
20. N. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, San Francisco, CA, 1996.
21. H. Lönn. Initial synchronization of TDMA communication in distributed real-time systems. In *The 19th International Conference on Distributed Computing Systems (ICDCS '99)*, Austin, TX, May 1999, pp. 370–379.
22. L. Pike. Formal verification of time-triggered systems. PhD thesis, Indiana University, Bloomington, IN, 2005.
23. W. Torres-Pomales, M. Malekpour, and P. Miner. ROBUS-2: A fault-tolerant broadcast communication system. NASA Technical Momerandum NASA/TM-2005-213540, NASA Langley Research Center, March 2005.
24. M. H. Azadmanesh and R. M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

25. P. M. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *Seventh Symposium on Reliable Distributed Systems*, Colombus, OH, 1988, pp. 93–100.
26. W. Steiner, M. Paulitsch, and H. Kopetz. The TTA's approach to resilience after transient upsets. *Real-Time Systems*, 32(3):213–233, 2006.
27. L. Pike, J. Maddalon, P. Miner, and A. Geser. Abstractions for fault-tolerant distributed system verification. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, Park City, Utah, Lecture Notes in Computer Science, 3223:257–270, 2004.
28. H. Pfeifer and F. von Henke. Modular formal analysis of the central guardian in the time-triggered architecture. *Reliability Engineering* & *System Safety*, 92(11):1538–1550, 2007.
29. H. Pfeifer. Formal analysis of fault-tolerant algorithms in the time-triggered architecture. PhD thesis, Ulm University, Germany, 2003.
30. F. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Cornell University, Ithaca, NY, August 1987.
31. N. Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Nijmegen, the Netherlands, Lecture Notes in Computer Science, 571:217–236, January 1992.
32. P. Miner. Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349, NASA Langley Research Center, January 1994.
33. D. Dolev, J. Halpern, and H. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 36(2):230–250, April 1986.
34. TTTech. Time-triggered protocol TTP/C High-Level specification document, Protocol Version 1.1. http://www.tttech.com/technology/specification.htm, 2003.
35. J. Rushby. Verification diagrams revisited: disjunctive invariants for easy verification. In E. Emerson and A. Sistla, Editors, *Computer-Aided Verification (CAV 2000)*, Chicago, IL, Lecture Notes in Computer Science, 1855:508–520, July 2000.
36. J. Rushby. An overview of formal verification for the time-triggered architecture. In W. Damm and E.-R. Olderog, Editors, *Proceedings of the Seventh International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Oldenburg, Germany. Lecture Notes in Computer Science, 2469:83–105, September 2002.
37. K. L. McMillan. Circular compositional reasoning about liveness. In L. Pierre and T. Kropf, Editors, *Correct Hardware Design and Verification Methods*, Bad Herrenalb, Germany, Lecture Notes in Computer Science, 1703:342–345, 1999.
38. R. Nieuwenhuis and A. Oliveras. Decision procedures for SAT, SAT modulo theories and beyond. The Barcelogictools. (invited paper). In G. Sutcliffe and A. Voronkov, Editors, *12h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, Montego Bay, Jamaica, Lecture Notes in Computer Science, 3835:23–46, 2005.
39. C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. Peled, Editors, *Computer Aided Verification: 16th*

International Conference, Vol. 3114 of Lecture Notes in Computer Science, Springer, 2004, pp. 515–518.

40. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 system. In Robert Nieuwenhuis, editor, CADE, *Lecture Notes in Computer Science*, 3632:315–321, 2005.
41. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, Editors, Seattle, WA, *CAV*, *Lecture Notes in Computer Science*, 4144:81–94, 2006.
42. L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE)*, Bremen, Germany, *Lecture Notes in Computer Science*, 4603: 2007.
43. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In M. Zelkowitz, editor, *Advances in Computers*, Vol. 58, Chapter 3. Academic Press, New York, 2003.
44. L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, Boston, MA, *Lecture Notes in Computer Science*, 3114:496–500, July 2004.
45. W. Torres-Pomales, M. R. Malekpour, and P. S. Miner. Design of the protocol processor for the ROBUS-2 communication system. NASA Technical Momerandum NASA/TM-2005-213934, NASA Langley Research Center, November 2005.
46. T. In der Rieden, D. Leinenbach, and W. J. Paul. Towards the pervasive verification of automotive systems. In D. Borrione and W. J. Paul, Editors, *CHARME*, Saarbrücken, Germany, *Lecture Notes in Computer Science*, 3725:3–4, 2005.

Index

Electrical Engineering

Automotive Embedded Systems Handbook

Highlighting requirements, technologies, and business models, the **Automotive Embedded Systems Handbook** provides a comprehensive overview of existing and future automotive electronic systems. It presents state-of-the-art methodological and technical solutions in the areas of in-vehicle architectures, multipartner development processes, software engineering methods, embedded communications, and safety and dependability assessment.

Divided into four parts, the book begins with an introduction to the design constraints of automotive-embedded systems. It also examines AUTOSAR as the emerging de facto standard and looks at how key technologies, such as sensors and wireless networks, will facilitate the conception of partially and fully autonomous vehicles. The next section focuses on networks and protocols, including CAN, LIN, FlexRay, and TTCAN. The third part explores the design processes of electronic embedded systems, along with new design methodologies, such as the virtual platform. The final section presents validation and verification techniques relating to safety issues.

Providing domain-specific solutions to various technical challenges, this handbook serves as a reliable, complete, and well-documented source of information on automotive embedded systems.

Features

- Presents a state-of-the-art account of the development processes, techniques, and tools of computer-based functions embedded in vehicles
- Offers balanced viewpoints from industry and academic experts, car manufacturers, and suppliers
- Outlines the features and functioning schemes of CAN, LIN, FlexRay, and TTCAN protocols
- Addresses the main problems in the design of automotive embedded systems, including time-to-market, variability, and response times
- Explains the classification-tree method, test scenario selection approaches, black-box/white-box testing processes, and fault-injection and monitoring techniques

TTTech The image on the front cover is provided by TTTech Automotive. TTTech Automotive offers products and engineering to bring FlexRay into commercial production.

8026

CRC Press
Taylor & Francis Group
an informa business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

ISBN: 978-0-8493-8026-6
90000
9 780849 380266