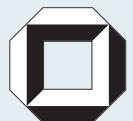
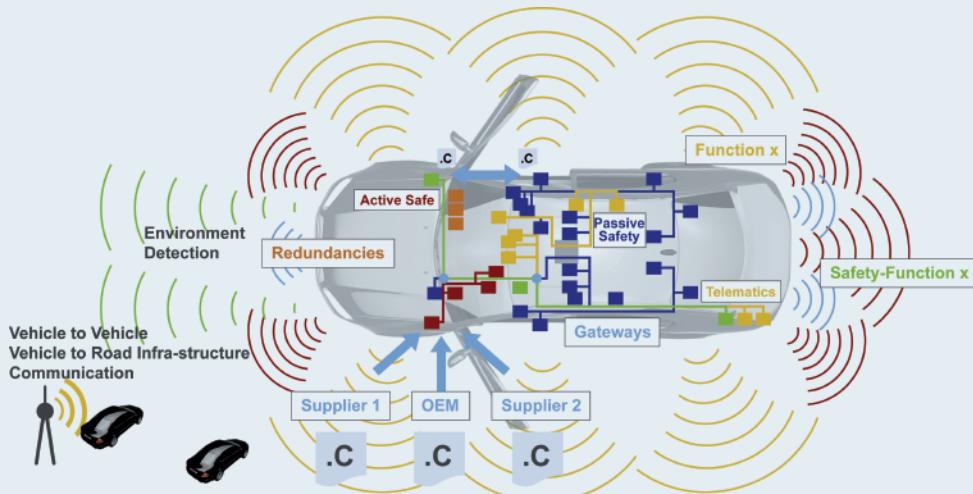


Xi Chen

Requirements and concepts for future automotive electronic architectures from the view of integrated safety

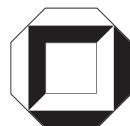


Xi Chen

Requirements and concepts for future automotive electronic architectures from the view of integrated safety

Requirements and concepts for future automotive electronic architectures from the view of integrated safety

von
Xi Chen



universitätsverlag karlsruhe

Dissertation, Universität Karlsruhe (TH)
Fakultät für Elektrotechnik und Informationstechnik, 2008

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de



Dieses Werk ist unter folgender Creative Commons-Lizenz
lizenziert: <http://creativecommons.org/licenses/by-nc-nd/2.0/de/>

Universitätsverlag Karlsruhe 2008
Print on Demand

ISBN: 978-3-86644-225-2

Requirements and concepts for future automotive electronic architectures from the view of integrated safety

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für

Elektrotechnik und Informationstechnik

der Universität Fridericana Karlsruhe (TH)

genehmigte

DISSERTATION

von

Diplom-Ingenieur Xi CHEN, M. Sc.

geboren in Wuhan

aus Schwäbisch Gmünd

Tag der mündlichen Prüfung: 07.02.2008

Hauptreferent: Prof. Dr.-Ing. K. D. Müller-Glaser

Korreferent: Prof. Dr.-Ing. G. Trommer

Karlsruhe, 09.01.2008

Acknowledgement

The presented work here was written during my work as a research associate from 2004 to 2007 by DaimlerChrysler (today Daimler AG) Group Research and Advanced Engineering in the department of Electric/Electronic Architecture.

I would like to sincerely express my gratitude to the supervisor of my dissertation, Mr. Prof. Dr.-Ing. Müller-Glaser, for his continuous invaluable guidance, encouragement and support. His vision and enthusiasm in advanced engineering research inspired and motivated me throughout my pursuit of the degree. I appreciate his broad knowledge, originality and insight in many areas. Working with him was a wonderful experience, and I have learned a lot from it. Many thanks to Professor Dr.-Ing. Trommer for his second expertise on this thesis.

Many thanks to Mr. Dr. Hedenetz, as the team leader of E/E-architecture of safety electronics, he supervised the technical work and gave me critical but very valuable comments on a draft of this work. Special thanks to Ms. Dr. Lauer for her support and for providing resources to perform the case study of this thesis. Acknowledgements here also to all my former colleagues in the department REI/EC at Daimler, for their interest in my work and for the professional discussions on this topic.

Also thanks to my project colleagues in WP1, WP2, WP4 and WP5 from EASIS industry consortium. They offered me many useful instructions and professional discussions. I also want to thank all my students, who helped me to build up the case study during their internship and master thesis.

I am forever indebted to my parents, Jianwei Chen, Gengxin Chen and my grandmother, Shunying Qing, for their great love, unselfish support, endless patience and strong encouragement through my life, especially the 9 years study and work in Germany, and my girlfriend, Xi Yu, for her patience and carefulness to read through my dissertation. I would dedicate all my achievements to them who I love so much.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als Ingenieur von 2004 bis 2007 bei DaimlerChrysler (heute Daimler AG) Group Research and Advanced Engineering in der Abteilung der Elektronik/Elektrik Architektur.

Mein besonderer Dank gilt

Herrn Prof. Dr.-Ing. Müller-Glaser für die Betreuung dieser Arbeit sowie die Unterstützung und Ermutigung zur Vollendung dieser Arbeit, seine Vision und Begeisterung in die Vorentwicklung und Forschung hat mich während der Arbeit ständig motiviert.

Herrn Prof. Dr.-Ing. Trommer für die Übernahme des 1. Mitberichts,

Allen Kolleginnen und Kollegen bei der Daimler Abteilung REI/EC für die gute Zusammenarbeit und hilfsbereite Unterstützung. Besonders möchte ich mich bei Herrn Dr. Hedenetz, als Teamleiter der Fahrzeug E/E-Architektur der Sicherheitselektronik, für die technische Bereuung und Korrekturlesen der Arbeit und Frau Dr. Lauer für die Unterstützung beim Aufbau des Validators bedanken.

Den Kolleginnen und Kollegen von WP1, WP2, WP4 und WP5 im EASIS-Konsortium für die erfolgreiche Zusammenarbeit und professionelle Diskussion. Den Studenten, die im Rahmen von Studien- und Diplomarbeiten zum Gelingen dieser Arbeit beigetragen haben, damit die vorgestellten Konzepte nicht nur Ideen blieben, sondern auch verwirklicht werden konnten,

Meinen Eltern Jianwei Chen, Gengxin Chen und meiner Großmutter Shunying Qing für ihre langjährige Unterstützung, unerschöpfliche Geduld insbesondere während meiner gesamten neunjährigen Studienzeit und Promotion in Deutschland, und meiner Freundin, Xu Yu, für ihre Geduld und Sorgfältigkeit bei dem Korrekturlesen meiner Dissertation.



Xi Chen

Schwäbisch Gmünd, im Dezember 2007

Abbreviation list

ABS	Anti-lock Braking System
AC	Aliveness Counter
ACC	Adaptive Cruise Control
ADC	Analog Digital Converter
API	Application Programming Interface
ARC	Arrival Rate Counter
AS	Activation Status
ASAM	Association for Standardization of Automation- and Measuring Systems
ASIC	Application Specific Integrated Circuit
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
BDHA	Basic Design Hazard Analysis
BMS	Battery Management System
BSW	Basic Software
CAN	Controller Area Network
CBC	Common Body Controller
CCA	Cycle Counter for Aliveness
CCAR	Cycle Counter for Arrival Rate
CCF	Common Cause Failures
CFC	Control Flow Checking
CFG	Control Flow Graph
CGW	Central Gateway
CMF	Common Mode Failures
COM	Communication
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DC	Diagnostic Coverage
DLL	Dynamic Link Library
DSC	Digital Signal Controller
E/E	Electric/Electronic
EASIS	Electronic Architecture System Engineering for Integrated Safety Systems

EAST-EEA	Electronic Architecture and Software Technology – Electronic Embedded Architecture
ECM	Engine Control Module
ECU	Electronic Control Unit
EDC	Error Detection Code
EEP	EASIS Engineering Process
EEPROM	Electrically Erasable Programmable only memory
E-Gas	Engine management system of gasoline and diesel engines
EMC	Electronic Magnetic Compatibility
EMS	Energy Management System
EPS	Electronic Power Steering
ESP	Electronic Stability Program
EU	European Union
EUS	Energy Uncoupling System
FAA	Functional Analysis Architecture
FDA	Functional Design Architecture
FHA	FAA Hazard Analysis
FMEA	Failure Mode and Effect Analysis
FMEDA	Fault Mode and Effect Diagnosis Analysis
FMF	Fault Management Framework
FOU	Fail Operational Unit
FPGA	Field Programmable Gate Array
FSM	Fault State Manager
FSU	Fail Silent Unit
FT	Fault Tolerance
FTA	Fault Tree Analysis
FTCom	Fault Tolerant Communication
FTDMA	Flexible Time Division Multiple Access
FTU	Fault Treatment Unit
HA	Hardware Architecture
HGA	Hazard Graph Analysis
HiL	Hardware in the Loop
HIS	Deutsches Automobilkonsortium - Hersteller Initiative Software

HU	Head Unit
HW	Hardware
I/O	Input/Output
IC	Instrument Cluster
ID	Identity
ISR	Interrupt Service Routine
ISS	Integrated Safety System(s)
ISS EP	ISS Engineering Process
IT	Information Technology
LIN	Local Interconnect Network
MCU	Micro Controller UNIT
MiL	Model in the Loop
MMU	Memory Management Unit
MPU	Memory Protection Unit
N/A	Not available or Not applicable
NA	Not Applicable
NM	Network Management
NVM	Non volatile memory
OEM	Original Equipment Manufacturer
OIL	OSEK Implementation Language
ORC	Occupant Restraint Controller
OS	Operating System
OSEK, OSEK/VDX	„Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug“, Open Systems and the Corresponding Interfaces for Automotive Electronics
PFC	Program Flow Checking
PFH	Probability of one dangerous Failure per Hour
PHA	Preliminary Hazard Analysis
PWM	Pulse Width Modulation
QoS	Quality of Service
RAM	Random Access Memory
RBTL	Reversible Belt Tensioner Left
RBTR	Reversible Belt Tensioner Right

RCP	Rapid Control Prototyping
ROM	Read-Only Memory
RTE	Run Time Environment
RTI	Real-Time Interface
RTOS	Real-Time Operating System
SBC	Sensotronic Brake Control
SC	Sensor Cluster
SER	Single Error Region
SFF	Safe Failure Fraction
SIL/ASIL	Safety Integrity Level / Automotive Safety Integrity Level
SOP	Start of Production
SotA	State of the art
SPC	Suspension Controller
SRS	Safety Requirements Specification
SSM	Standard Software Module
SW	Software
SW-C	SoftWare Component
SW-Cs	Software Components
SWM	Steering Wheel Module
SW-Watchdog	SoftWare Watchdog
TCM	Transmission Control Module
TSIU	Task State Indication Unit
TDMA	Time division multiple access
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
V&V	Verification and Validation
VFB	Virtual Functional Bus
WCET	Worst Case Execution Time
WSM	Wheel Steering Module

Abstract

Integrated Safety System, as one of the most promising technologies of the automotive safety systems, integrates passive and active safety system, together with the interaction of cabin, chassis and powertrain electronics cross the domain board and communication networks. It provides the largest potential for future innovative safety applications. The development of ISS, compared with the current in-vehicle safety systems, puts forward more challenging requirements for the design, development, prototyping and validation process. In order to manage the complexity and fulfill high dependability requirements, new architecture concepts, engineering process and tool chains for the Integrated Safety Systems are required.

In this dissertation, requirements of future Integrated Safety Systems are identified firstly with a delta-analysis. Based on these requirements, concepts of the electronic architecture for the Integrated Safety Systems are developed as a cooperative approach of engineering process, dependable hardware architecture and software platform. In this safety justified development process, with the methodology of virtual front-loading, a distributed rapid prototyping based on the principle of early integration and validation with advanced simulation environment is suggested. The strict and well defined development steps specified here enables the correct-by-construction for the development of ISS. Dependable hardware architecture for ISS is discussed in a top-down methodology, starting with concepts to build up overall in-vehicle topology, guidelines to distribute ISS-applications, design of communication systems and concepts of fault-tolerant ECU hardware architecture are introduced. To conclude this topic, some of the most important use cases to the concepts are demonstrated. Strategy of standardized software platform has been proven to be one of the most effective levers to manage the complexity of future automotive software. Facing the challenges of higher fault tolerance towards fail-operational and increasing density of application software components on one ECU, dependability software services for the fault-tolerant communication with redundancy, end-to-end CRC and Agreement Protocol, time partitioning and space partitioning for the integration of applications, Fault management, fault treatment of dynamic reconfiguration and gateway services are designed and integrated in a layered software topology.

For the practical evaluation and validation of the concepts introduced here, a hardware-in-the-loop validator based on the Steer-by-Wire technologies was built up. On top of this validator, serial-near ISS-applications are integrated and distributed on different ECUs. The dependable software and hardware concepts are designed, prototyped and tested with fault injection on various platforms. The ISS Engineering Process and the tool-chains for the prototyping have demonstrated their viability for the development of complex ISS-applications during the validation process as well.

Zusammenfassung

Motivation:

Integrierte Sicherheitssysteme versprechen aufgrund der Interaktion und Synergieeffekten von Passiv- und Aktivsicherheitssystemen mit Innenraum, Fahrwerk und Antriebstrang, das größte Potential für die Weiterentwicklung von zukünftigen Fahrzeugsicherheitselektroniken. Die Entwicklung der integrierten Sicherheitssysteme, stellen im Vergleich zu den herkömmlichen Fahrzeugsicherheitssystemen höhere Anforderungen an den Entwurfs-, Entwicklungs-, Musteraufbau- und Validierungsprozess. Um die Komplexität und die höheren Anforderungen zu beherrschen, sind neue Architekturkonzepte, Entwicklungsprozesse und Werkzeugkonzepte für die integrierten Sicherheitssysteme notwendig.

Die Arbeit:

In der vorliegenden Arbeit werden die Anforderungen der zukünftigen integrierten Sicherheitssysteme analysiert. Auf dieser Basis wurden Konzepte der Elektronikarchitektur der integrierten Sicherheitssysteme ausgearbeitet, dabei wurde ein Ansatz gewählt, welcher den Entwicklungsprozess, die Hardwarearchitektur und die Softwareplattform mit einschließt. In dem sicherheitsgerechten Entwicklungsprozess, mit der Methodik der virtuellen Front-loading, wird ein verteiltes „Rapid-Prototyping“ auf der Basis der Frühintegration und Validierung mit der Simulationsumgebung vorgeschlagen. Die strikten und vordefinierten Entwicklungsschritte ermöglichen die „correctness by construction“ (frühzeitige Konzeptabsicherung) bei der Entwicklung der ISS. Verlässliche Hardwarearchitektur für ISS wurde nach dem „top-down“ Ansatz entwickelt, beginnend mit dem Konzept zum Aufbau der Fahrzeugtopologie des Kommunikationsnetzes, Richtlinien zur Verteilung und Abbildung der ISS-Applikationen, Entwurf des Kommunikationssystems und Konzepte der fehlertoleranten ECU HW-Architektur werden vorgestellt. Anschließend werden ein paar wichtige Fallstudien zu Demonstrationszwecken der Konzepte erläutert. Die Strategie der standardisierten Softwareplattform ist eine der wichtigsten Ansätze um die Komplexität der zukünftigen Automobilsoftware zu beherrschen. Um den Herausforderungen der höheren Sicherheitsanforderungen der fail-operational Systeme und der steigenden Dichte von Applikationssoftware auf einem Steuergerät gerecht zu werden, wurden fehlertolerante Softwareservices zur Unterstützung der Redundanz, verlässliche Kommunikation, Ende-zu-Ende Applikations-CRC Übereinstimmungsprotokoll, zeitliche und räumliche Auftrennung bei der Applikationsintegration, Fehlermanagement und Fehlerbehandlung mit dynamischer Rekonfiguration entworfen und in einer schichtenorientierten Softwarearchitektur integriert.

Validierung der Konzepte:

Zum praktischen Nachweis des erstellten Konzeptes wurde ein auf Steer-by-Wire basierender Hardware-in-the-Loop Validator entwickelt. Auf oberster Ebene des Validators wurden die seriennahen ISS-Applikationen integriert und auf verschiedenen ECU Plattformen verteilt. Die davon abhängigen Software- und Hardwareanteile wurden entworfen, prototypisch implementiert und mit Fehlerinjektion auf verschiedenen Plattformen getestet. Die Stärke des ISS-Entwicklungsprozesses und der dazugehörenden Werkzeugkette wurde beim Prototyping und bei der Validierung der komplexen ISS-Applikationen demonstriert.

Table of contents

<i>Acknowledgement</i>	ix
<i>Danksagung</i>	x
<i>Abbreviation list</i>	xi
<i>Abstract</i>	xv
<i>Zusammenfassung</i>	xvi
<i>Table of contents</i>	xvii
1 <i>Introduction</i>	1
1.1 Short introduction to future automotive safety systems	1
1.2 Motivation for the work	3
1.3 Overview of the dissertation	4
2 <i>Basic concepts of dependability</i>	7
2.1 Definition of dependability	7
2.2 Introduction to fault tolerance	10
2.2.1 Fault diagnosis/detection	10
2.2.2 Fault description	11
2.2.3 Fault treatment	14
3 <i>State of the art: Automotive electronic architectures</i>	17
3.1 Building blocks of automotive electronic architectures	17
3.2 Requirements for future automotive safety systems.....	18
3.3 State of the art: Hardware architecture for automotive electronics.....	20
3.3.1 Requirements for hardware of safety electronic systems	20
3.3.2 In-vehicle system topology	20
3.3.3 ECU hardware architecture	21
3.3.4 Automotive communication systems	27
3.4 State of the art: Software architecture for automotive electronics	31
3.4.1 Requirements for software in safety electronic systems.....	31
3.4.2 ECU software architecture	32
3.4.3 Automotive software services	36
3.4.4 Conclusion for automotive software	37
3.5 State of the art: Development process for automotive electronic systems	38
3.5.1 Requirements for the development process of safety electronic systems	38
3.5.2 State of the art in development processes	38
3.5.3 Conclusion for development process	50
4 <i>Assessment of state of the art approaches and conclusion for challenges</i>	51
5 <i>Introduction to EASIS project and EASIS approaches</i>	53
6 <i>Fault type and fault hypothesis for ISS</i>	57

6.1	Fault hypothesis – a major design step	57
6.2	Fault hypothesis of hardware in ISS	58
6.3	Fault hypothesis of software in ISS	62
6.3.1	Software timing faults	63
6.3.2	Communication between SW-components	63
6.3.3	Concurrent resource access	64
7	<i>Engineering Process of Integrated Safety System</i>	65
7.1	Introduction to the ISS Engineering Process	65
7.2	Development steps of ISS Engineering Process	70
7.2.1	Part 1: Initial of requirement engineering (specify preliminary requirements)	71
7.2.2	Part 2: Development of Functional Analysis Architecture (FAA Model)	73
7.2.3	Part 3: Development of hardware architecture	78
7.2.4	Part 4: Development of Functional Design Architecture (FDA)	79
7.2.5	Part 5: Refinement and validation of FDA model with SiL-test	84
7.2.6	Part 6: Hazard analysis and validation of FDA model with HiL-test	89
7.2.7	Association of ISS Engineering Process with ISO26262	91
7.3	Tool chains for the development of ISS	96
7.3.1	Software tools	97
7.3.2	Hardware prototyping platforms	100
7.4	The ISS Engineering Process under challenges	101
7.4.1	View from side of OEM	101
7.4.2	View from side of supplier	103
7.4.3	Distributed cooperation between OEM and suppliers	104
8	<i>Hardware architectures for the Integrated Safety System</i>	105
8.1	Concepts of the system topologies	105
8.1.1	Future frameworks and design guidelines of system topologies	106
8.1.2	Distribution of ISS applications to the system topology	108
8.2	Design concepts of the communication systems	111
8.3	Concepts of ECU hardware architectures	113
8.3.1	Dependable architecture driven by the safety integrity requirements	113
8.3.2	Monitoring of sensor and actuator components	117
8.3.3	Memory protection	118
8.3.4	Hardware watchdog monitoring	119
8.4	Design use-cases of ISS hardware architectures	120
8.4.1	Distribution of ISS applications to vehicle domains	120
8.4.2	ISS system topologies and communication systems	122
8.4.3	ECU hardware architecture for ISS	125
8.5	Conclusion of hardware architecture framework	126
9	<i>Software platform for the Integrated Safety Systems</i>	127
9.1	Trends of software platform – a benchmark with IT-industry	127
9.2	Concepts of dependability software architecture	128

9.3	Concepts of the dependability software services	131
9.3.1	Dependability services for ISS communication	131
9.3.2	Dependability services for the integration of applications on one HW-platform.....	146
9.3.3	Dependability services of fault treatment.....	162
9.3.4	Dependability software services for gateway	174
9.4	Configuration of dependability software services	175
9.5	Conclusion of the dependable software platform	177
10	<i>Prototyping and validation of the concepts</i>	179
10.1	Introduction to the architecture validator	179
10.1.1	Prototyping approaches and validation process	179
10.1.2	Architecture design of the validator	180
10.2	Validation of hardware architectures for ISS	182
10.3	Validation of dependability software platform and services.....	184
10.3.1	Prototyping and validation of Agreement Protocol	185
10.3.2	Prototyping and validation of Software Watchdog.....	187
10.3.3	Prototyping and validation of Fault Management Framework.....	189
10.4	Validation of the ISS Engineering Process.....	191
10.5	Evaluation and optimization of the concepts	194
10.6	Experience and findings from the prototyping and validation	195
11	<i>Conclusion of the results, discussion and outlook</i>	197
11.1	Conclusion and implication of dependability architecture framework	197
11.2	Outlook for the future work	199
Appendix 1:	<i>Implementation details of the validator</i>	200
Appendix 1.1	Validation of hardware architectures for ISS	201
Appendix 1.2	Validation of dependability software platform and services.....	203
Appendix 1.2.1	Prototyping and validation of Agreement Protocol	203
Appendix 1.2.2	Prototyping and validation of Software Watchdog	205
Appendix 1.2.3	Prototyping and validation of Fault Management Framework	207
Appendix 2:	<i>Mathematic derivation of the communication overhead of agreement protocol</i>	211
Appendix 3:	<i>Glossary</i>	212
Application Software Component	212	
Basic Software Module	212	
Compositionality.....	212	
Software Configuration	212	
Control Flow	212	
Event	212	
Fail-Degraded	212	
Fail-Operational	213	
Fail-Safe	213	
Fail-Silent	213	

Table of contents

Fault Containment	213
Front Loading	213
Redundancy	213
<i>Appendix 4: List of figures</i>	214
<i>Appendix 5: Literature index</i>	218
<i>Appendix 6: Author's biography</i>	225
<i>Appendix 7: Lebenslauf</i>	226

1 Introduction

1.1 Short introduction to future automotive safety systems

Nowadays safety and comfort are two of the most important requirements and innovation factors for automotive OEMs and suppliers. In order to increase the safety of passengers within a vehicle and for the surrounding environment, more and more electronic systems are used in the automotive safety domain.

In the safety domain there are traditionally two safety systems: the active safety system and the passive safety system.

The following components belong to passive safety components: airbags, collapsible zones, bumpers and safety belts, etc. They help to reduce the negative effects of a traffic accident by means of the physical absorption of crash energy. On the other hand, active safety components include ABS, ESP, brake assistance and lane keeping/change assistance, which influence the powertrain, drive train and chassis system directly, in order to avoid accidents. Current active safety systems predominately directly influence the brake system only, while future active safety systems will also integrate the steering and directional stability track-holding system.

The passive safety system has already reached a relatively mature standard, in which innovation potential is almost exhausted and significant improvements are unlikely to be reached in the near future. Future safety enhancements will mainly have to rely on active safety systems to avoid accidents before they can occur and the so-called “Integrated Safety System” as a combination of both active and passive safety systems, as well as other automotive domains. An overview of the safety level reached and forecasted future development in the passive and active safety domains [GDi02] is shown in Figure 1-1, in which the potential development of ISS is demonstrated as well.

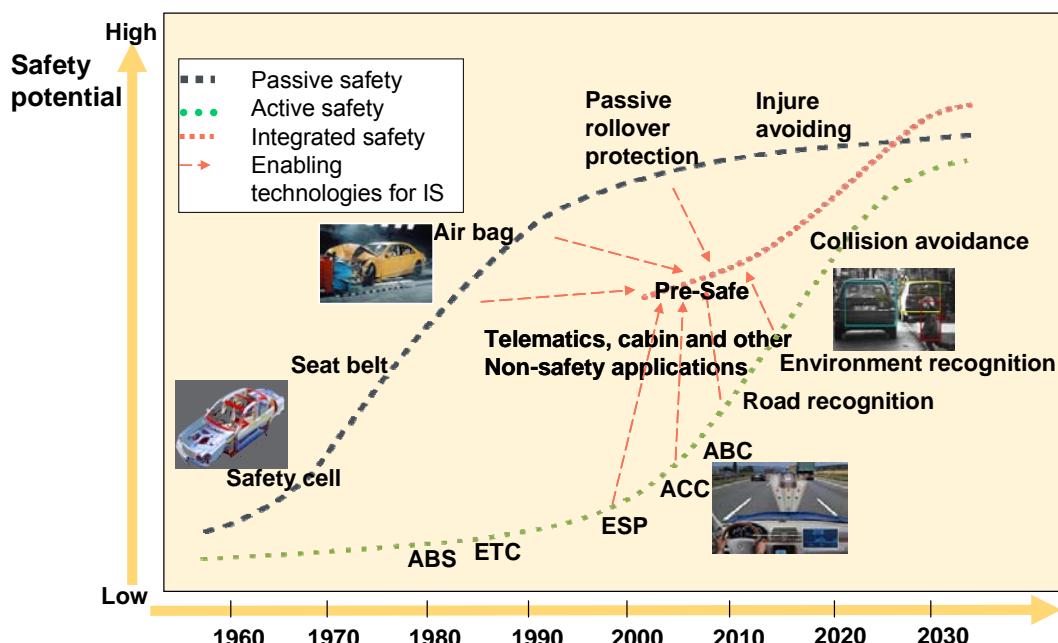


Figure 1-1: Development of active, passive and integrated safety electronic systems

The European Commission's transport policy has set new targets for 2010 with respect to road safety, as shown in Figure 1-2, where a dramatic reduction in road fatalities is expected by 2010. The traditional approach with isolated consideration for passive safety electronics, active safety electronics and other automotive domains does not provide enough potential to reach the ambitious aims of European Commission transport policy. The Integrated Safety System is the only realistic approach to the targets set by EU [EUC01].

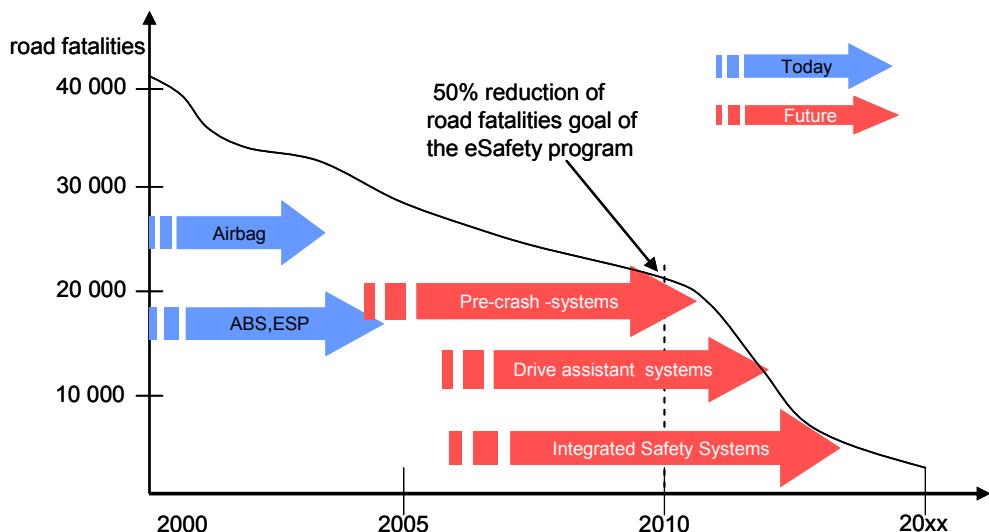


Figure 1-2: Targets set by EU of 50% reduction in the road fatalities before 2010

An Integrated Safety System is a composition of functions that enhance the level of safety not only for the passengers but also for the environment such as pedestrian, which spreads across domain borders, including passive safety components such as airbags, collapsible zones, bumpers, safety belts, etc. and active safety functions such as ABS, ESP, brake assistance, lane change assistance, etc.

The special aspects of ISS-applications compared with the traditional stand-alone safety relevant system can be summarized as follows:

- An ISS-application can spread across domain boarders, e.g., the Pre Crash System closes side windows / sunroof.
- An ISS-application can consist of functions from different domains with different safety integrity levels and safety requirements.
- An ISS-application makes use of a plethora of components, which themselves are not necessarily safety relevant (e.g. sliding roof / sunroof) in itself, thus, comfort components may inherit a high safety level.

With the interaction and integration of safety relevant / non-safety relevant applications from different domains and data sharing from environment sensors and telematics domains, the same data will not need to be measured in different domains, likewise, new applications can be created by using data which is not available from the original domain. The main benefits of Integrated Safety are listed as following:

- Information from all domains can be combined to provide a better view about the state of the vehicle and its surroundings, thereby forming a better basis for decisions taken by safety systems;
- The vehicle can be controlled in a more integrated way as control actions can be coordinated across domains.

A good example of an ISS-Function, combining passive and active safety electronics is the Pre-Safe system [ATZ05] from Mercedes-Benz (first introduced in the S-Class 2002), which detects an unavoidable crash and makes preparations to reduce the severity of the accident. The activation trigger for the Pre-Safe system is the velocity of brake pedal, vehicle velocity and plausibility of on-board electronics. Measures taken by the Pre-Safe system include activation of a reversible belt restraining system, adjusting of seat position and closing of side windows and sunroof.

1.2 Motivation for the work

As discussed in the subsection above, many potential innovations in safety electronics lie in the integration of passive safety electronics, active safety electronics and other domains. By means of sensor data fusion and information flow beyond domain borders, more new safety applications could be designed. From a technical point of view, however, current safety systems are mainly stand-alone systems with a limited degree of interdependency, which can not fulfill the requirements of ISS. An integrated approach for vehicle safety systems, in particular the cooperation between the passive and active safety domains, as well as cabin, chassis and telematics domains, offers a solution to enable innovation and improve safety for both passengers and the environment. This includes the combination of active and passive safety systems, where actions will address all phases of a crash, including pre-crash and post-crash.

Dependability has always been the most important requirement for automotive safety relevant applications. For customers, it is not important whether the safety applications are implemented in traditional mechanics or with innovative electronics, but they do expect the new safety features to be at least as reliable as traditional systems.

From a hardware perspective, a safety system with redundant backups is the state of the art approach for a highly dependable system, while taking into consideration the difference in general conditions, as opposed to those specific to the automotive industry. For the future development of safety systems, however, the mechanical backup can disappear. That is to say, compared with the current automotive safety systems, which are only fail-safe or fail-silent, future integrated safety systems should be fail-operational or fail-degraded.

With the increasing number of ECUs and complexity both in hardware and software, it is a great challenge to implement complex, distributed applications, with shorter development cycles while still keeping the dependability of system at a high level. One solution is the standardization of the ECU middleware based E/E-platform to increase the modularity of software components. This

trend can now be observed in different consortiums of OEMs and suppliers. With the improvement in reusability, the test depth of system components can be intensified and development time will be shortened because of the unified interface. In the automotive safety domain, the standardized ECU middleware should also support fault-tolerance mechanisms, due to the higher requirement for dependability. A fault-tolerant middleware architecture, adapted to the specific requirements of the automotive safety domain, is not available yet.

The development process for safety relevant electronic systems, compared with the general development process of automotive electronic systems, has to be adapted due to the high dependability requirements. An example of such safety measure is the introduction of preliminary hazard and safety analysis, system hazard analysis and binding to the appreciate development steps. With the derived safety integrity level from safety analysis, the development process for software should be configured as required in industry safety norms. For the development of an Integrated Safety System, the cooperation of sub-systems and their integration into the overall system is of vital importance, because even if all the sub-systems appear to work, the whole system can still fail. An early validation of the system specification is not yet fully supported by the current development process.

1.3 Overview of the dissertation

This dissertation is organized as follows:

After a short overview of current and future automotive safety systems in Subsection 1.1, motivation for the dissertation is explained in Subsection 1.2. Following an introduction to the basic terms of dependability and fault tolerance in Chapter 2, the state of the art in overall automotive electronic architecture, including an introduction to dependability, system topology, automotive software, hardware and communication systems considering safety relevant systems, their current and future requirements are explained in Chapter 3. The development process for generic electronic systems, special concepts and methods for automotive safety systems and an assessment of existing methods will be also introduced in this chapter as well.

In Chapter 4 the existing concepts, trends and current initial concepts in automotive electronic architecture and dependable electronics will be evaluated. From the challenges of ISS and known concepts, a delta analysis will be carried out. The problem is addressed from the different viewpoints of hardware, network topology, communication networks, software with FT-middleware and the development process.

In Chapter 5 EASIS (*Electronic Architecture and System Engineering for Integrated Safety Systems*) project and the approach is briefly introduced. The research work of this dissertation took place during the participation in this project.

As a basis of the design of dependability architecture framework, a dedicated for Integrated Safety System configured faulty hypothesis and fault types was carried out. The result of the analysis is specified in Chapter 6.

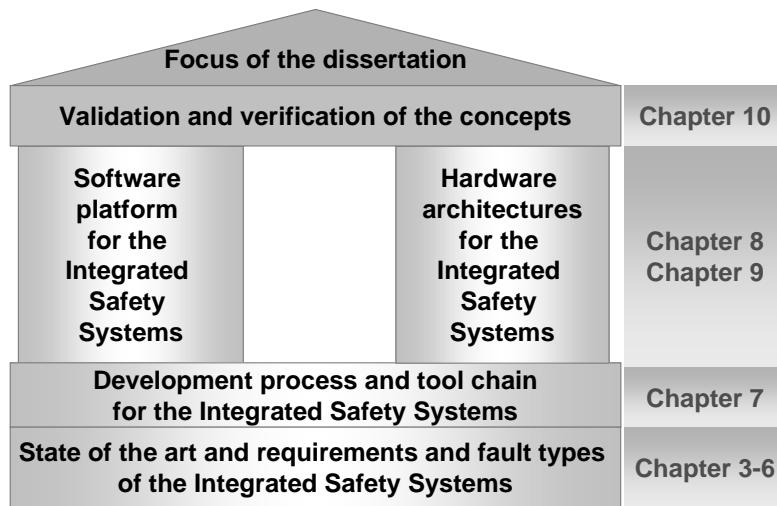


Figure 1-3: Structure of the focus in the dissertation

As depicted in Figure 1-3, emphasis of the dissertation lies in the Chapter 7 to Chapter 9. Chapter 7 introduces ISS Engineering Process for the prototyping and development of Integrated Safety System. The ISS Engineering Process takes the future development trends of safety hardware and software architecture and safety norm into consideration, follows two main principles: virtual front-loading and correct by construction approach.

Triggered by the quantitative requirements from safety norms, hardware architecture framework for the Integrated Safety System of system topologies, ECU hardware architecture and design use-cases are discussed in Chapter 8.

In Chapter 9, dependability software platform with dependability software services for ISS-communication, integration of applications with safety integrity level, fault management and gateway services are specified.

Prototyping and validation of the safety concepts proposed here are illustrated in Chapter 10. Chapter 11 concludes the work with experiences gained in the design and development of the ISS architecture framework and provides an outlook for future work. Last but not least, the appendixes include more implementation details of the validator, mathematic derivation, glossary, list of figures, literatures and author's biography.

2 Basic concepts of dependability

2.1 Definition of dependability

Since there is no common definition of dependability in existing literatures and different norms, in the following, a general definition of dependability is given with regard to the framework of this dissertation.

Dependability: Dependability is defined as the trustworthiness of a computer system, such that reliance can justifiably be placed on the service it delivers. This is a bundle of terms, as illustrated in Figure 2-1.

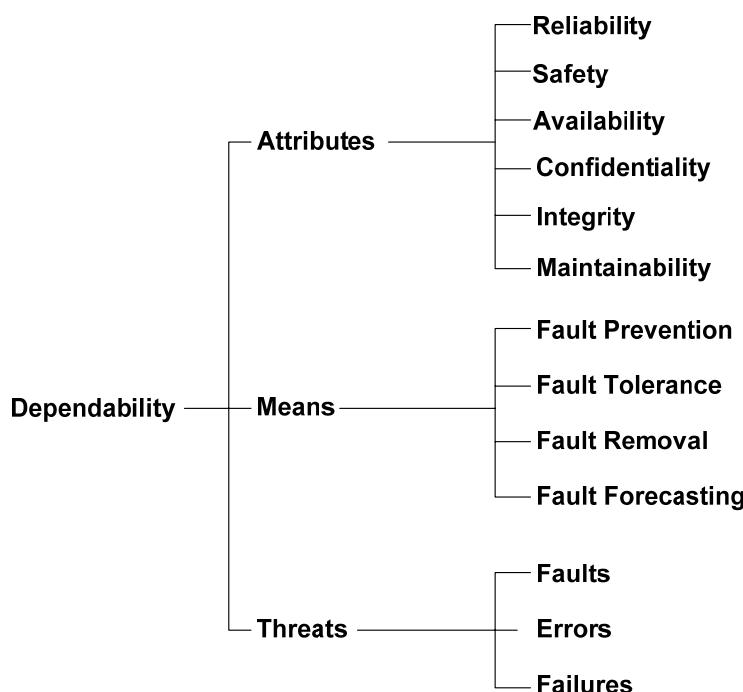


Figure 2-1: The dependability tree [Lap95]

The characteristic attributes of dependability in Figure 2-1 are explained as follows:

- **Availability** is the readiness for correct service – “Uptime” as a percentage.
- **Reliability** is the continuity of correct service, thus how likely it is that the system can complete a mission within a given duration.
- **Safety** is the absence of catastrophic consequences on the user(s) and environment. Two term concerning safety, the safety integrity and safety integrity level are explained as follows:
 - *Safety Integrity* defines the probability of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a specified period of time.

- *Safety Integrity Level* is used to describe different safety requirements. In the industry norm IEC 61508 [IEC01], SIL is specified as discrete levels of safety relevance, which require different approaches to be realized. Different safety applications involved with the same sub-systems, can have different SILs.
- **Confidentiality** is the absence of unauthorized disclosure of information.
- **Integrity** is the absence of improper system state alterations (Note: security involves malicious faults; confidentiality & integrity).
- **Maintainability** is the ability to undergo repairs and modifications.

The system dependability is a trade-off of the mentioned attributes here, e.g. the different monitoring mechanisms can improve the system safety but not necessarily the availability and maintainability since safest system is the system which doesn't work at all.

The threats of dependability are listed as follows and their relationship to the system is illustrated in Figure 2-2:

- **Fault** defines an abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. As shown in Figure 2-2, fault is the cause of a system failure.
- **Error** defines a discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. An example of an error is the occurrence of an incorrect bit caused by an equipment malfunction. Error is a system state that causes failure.
- **Failure** defines the termination of the ability of a system or functional unit to perform a required function. A failure in a sub-system can be a fault for a higher layer system. The latency time from fault to system failure is labeled as t_1 , t_2 and t_3 .

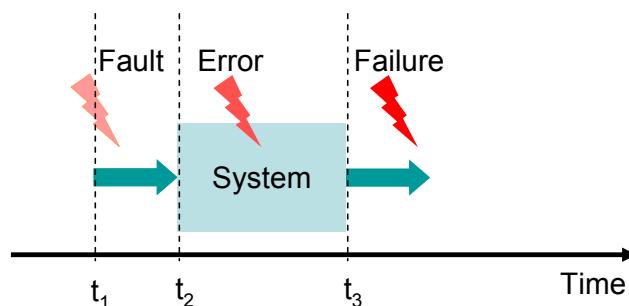


Figure 2-2: Difference between fault, error and failure

In case of failure, different strategies can be performed so as to avoid a hazard to the system, such that it continues as fail-silent, fail-safe or fail-degraded. The definition of such terms and their relationship are explained as follows:

The current semi X-by-Wire systems (e.g. SBC, EPS, E-Gas...) use a mechanical or directly cabled backup system, which exhibits the “fail-safe” and “fail-silent” characteristic.

- **Fail-silent** characterizes the property of a system or functional unit, where in the case of a fault, the output interfaces are disabled in such a way that no further outputs are made. Fail-silent is a special case of the fail-safe property.
- **Fail-safe** defines the property of a system or functional unit, where in case of a fault, the system or functional unit transits to a safe state.

For the future automotive safety systems without mechanical backup (real X-by-Wire systems), it is quite difficult or almost impossible to define a safe state for the vehicles on the road. Therefore, they must have a significantly higher dependability, which should be at least as high as the comparable mechanical system. This means that the vehicles should still offer a higher level of safety to the passenger, even if one or more sub-systems should fail. Thus fail-degraded or fail-operational systems are required.

- **Fail-degraded** defines the property of a system or functional unit, which has the ability to continue with intended degraded operation at its output interfaces, despite the presence of hardware or software faults, the “Limp home” functionality for ECU is an example of this (reduce torque to ensure arrival at home or service station).
- **Fail-operational** describes the property of a system or functional unit, which can continue normal operation at its output interfaces despite the presence of hardware or software faults or errors, for example in the braking system.

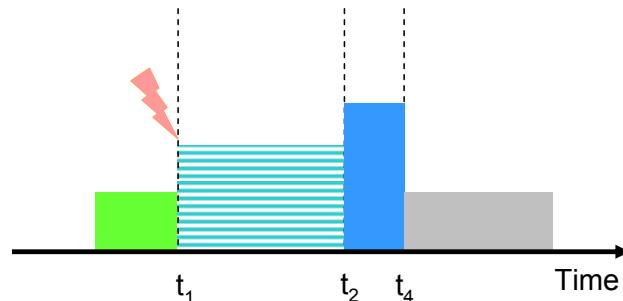


Figure 2-3: Timing requirement of fault tolerance

The timing requirement of fault tolerance is depicted in Figure 2-3. For $t \leq t_1$ the whole system is in a fault free state, $t_1 \leq t < t_2$ is an abnormal system state, in which the fault detection should take place when e.g. the monitoring threshold is exceeded. During the time period $t_2 \leq t < t_4$ a reconfiguration of the system as to the fault treatment should take place, in the time after t_4 ($t \geq t_4$) the whole system is in a reconfigured state and should exhibit a fail-safe fail-degraded or fail-operational behavior. For the sake of fault tolerance, $t_4 - t_1$ should be smaller than $t_3 - t_1$ in Figure 2-2, so that no failure should happen.

A relationship between the different operational states of an ECU, such as failure, normal, fail-operational and fail-safe (including fail-silent and fail-degraded) is illustrated in Figure 2-4.

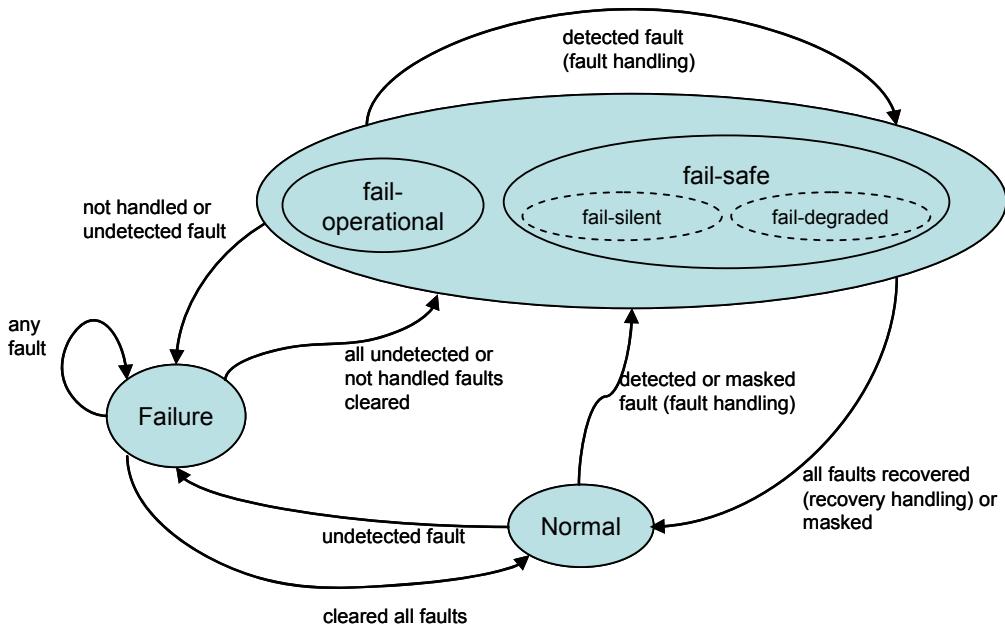


Figure 2-4: Relationship between different operational modes of an ECU

2.2 Introduction to fault tolerance

Various concepts are discussed in different literatures as to how to improve dependability, for examples by means of fault-prevention, fault-recovery, fault-tolerance and fault-forecasting, etc. Fault-tolerance is one of the most methodic of these concepts, which will be explained here in detail [Ech90].

Fault-tolerance describes the ability of a functional unit to continue to perform a required function in the presence of faults or errors. A fault tolerant system has to provide at least three activities:

- Fault diagnosis/detection
- Fault description
- Fault handling including fault isolation (damage assessment) and fault recovery

Fault tolerance removes the effects of the failure. With respect to dependability, this solution is always the best, although it may be the most expensive. In the following sub chapters the three activities/steps will be explained in detail.

2.2.1 Fault diagnosis/detection

One of the most important elements of fault-tolerance is fault diagnosis or fault detection. This describes the ability of a system, to detect and localize a particular abnormal state or state combination in a system or the environment, which has resulted in or will result in an error.

Fault diagnosis serves as the basic and the very first step, in order to avoid a fault in the global system, because measures to restrict and avoid faults in the components and system can only be carried out when unallowable states are detected.

As a conclusion, fault diagnosis can be divided into two steps. The first step is to detect the existence of a fault. The second is to localize the fault, where the error arises. This is important, in order to choose appropriate fault-handling mechanisms for the faulty part(s).

2.2.2 Fault description

In order to implement an efficient approach for fault detection, it is necessary to be aware of the potential faults. An appropriate monitoring mechanism can only be developed with the knowledge of which kinds of faults can occur at which components.

For this reason, it is necessary to define a set of potential faults, by analysis of the system and possible faults, which can be tailored in a top-down process to gain a so-called fault model. The probability of the respective faults and their hazard to the system as a whole will then be analyzed. Finally, the set of faults which can be tolerated, taking different constraints into consideration will be defined.

2.2.2.1 Fault classes

It is necessary to categorize the different faults into appreciate fault classes by analyzing the potential faults in the system. In this sub-chapter a number of basic fault classes will be specified.

There are three basic faults:

- System fault,
- Run time fault
- Other faults

System faults are faults which are embedded at the starting of the system due to conceptual problems. Run time faults, which occur during the running time, are the second most common type of fault. Run time faults can be caused by wearing faults, operating faults, coincident faults and malicious faults.

The third category of faults is other faults, which do not originate from the system nature or operational period, for example production faults or maintenance faults.

Faults can be divided further, according to their location:

- Faults in software
- Faults in hardware
- User operating faults

The final classification of faults is made according to the duration of the fault:

- Permanent faults: continuous until maintenance or other appreciate fault handling method is carried out.
- Temporary faults: last only for short time and disappear after a while.

Table 2-1 shows, as a conclusion, the faults discussed in this sub-chapter and their relationship to fault occasion, fault location and fault duration.

		software	hardware	user	permanent	temporary
system fault	specification faults	X	X		X	
	design fault	X	X		X	
	implementation fault	X			X	
	document fault	X	X		X	
runtime fault	wearing fault		X			
	operation fault			X		X
	interference fault		X			X
	coincident fault		X			X
	deliberate fault			X		X
other faults	production fault	X	X		X	X
	maintenance fault	X	X		X	X

Table 2-1: Fault classes [Ruh04]

Exclusively for distributed electronics, there are two main categories of faults [Cri91]:

- Non-specified behaviors during runtime
- Neglect of input signals and functionalities can be only restored after a restart

The first category can be divided into the following sub-categories:

- Omission fault
- Timing fault
- Response fault
- Byzantine fault

While the second category can be divided into following parts:

- Interval crash
- Partial-amnesia crash with partial history loss
- Amnesia crash with total history loss

In the following, the seven failure categories are specified in detail:

- *Omission faults*: Omission faults describe the situation when a required functionality is not performed within a certain time. A typical example is the communication between two ECUs. ECU1, according to the specification, should respond to ECU2 with a pre-defined signal. Because of interference in the communication channel the message is, however, lost or damaged. ECU2 can recognize faults by monitoring time out.
- *Timing faults*: In timing faults the required services are not delivered in the pre-defined time window. Such failure occurs very often when the communication channel is overloaded or the microprocessor is too busy.
- *Response fault*: when the required service can not be fulfilled as specified, for example faulty response/control of actuator or non-specified state change.
- *Byzantine fault*: One of the most difficult faults to handle is Byzantine fault. In this fault one "identical" message, from the point of view of the sender, will be sent out with different content to several receivers, so that the receivers, which are supposed to process identical information, have different input values. There are two different Byzantine faults: "Benign Byzantine Fault" and "Malicious Byzantine Fault". In a benign Byzantine fault a certain number of receivers can receive an identical, valid value and the rest receive nothing at all. With malicious Byzantine faults, different values will be received by the receivers.

- *Pause crash*: Pause crash is the crash with the lightest influence on the complete system, in which the functionalities of electronics resume work as normal.
- *Partial-amnesia crash with partial history loss*: compared with pause crash, only one part of the state information from before the crash remains for further operation, the rest are lost. Every ECU has some kind of history, that is to say, information regarding past actions, values and states. One example of partial data loss is a restarted ECU, which uses history information to compare computed state with real environment. A concrete example is the loss the last steering actuator angle of an EPS before the next ignition cycle.
- *Amnesia crash with total history loss* is unfortunately the most common case in automobile electronics, in which a continuation from the previously computed results is impossible.

2.2.2.2 Fault propagation behavior

When one component in the system exhibits a faulty behavior, a so called Domino Effect could occur, in which n to n+1 further components break down and this leads to a crash of the entire system. Generally speaking, there are two categories of fault propagation:

- Vertical fault propagation
- Horizontal fault propagation

The basis for analysis of fault propagation is the layered architecture model (for example [ISO7498-1]) of the electronics, a hierarchical architecture from hardware at the very bottom to the software application at the very top, as shown in Figure 2-5.

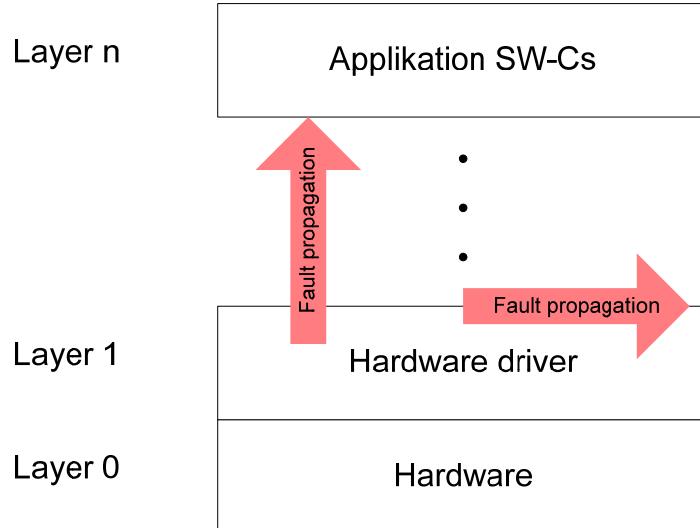


Figure 2-5: Layered architecture model of electronics

Fault propagation, which only results in other faults at the same layer, is called horizontal fault propagation. Fault propagation over the layer border is called vertical propagation.

2.2.2.3 Fault model, fault hypothesis and fault types

A fault model is a model, which describes the structure of a system, including its components and the fault possibilities (=malfunctions) of the individual components, in an at least qualitative way.

In the automobile industry there are various methods for analyzing the fault model:

- Dependability analysis
- Event Tree Analysis (ETA) und Fault Tree Analysis (FTA),
- Failure Mode and Effect Analysis (FMEA),
- Failure Mode, Effect and Diagnosis Analysis (FMEDA)
- Hazard analysis (HA)

A fault hypothesis denotes a claim about the set of faults (independent or related faults) and states an assumption about their types defined by fault number, source and temporal behavior, which are tolerated by a system within a certain time interval. [Kru98]

Detailed discussion concerning fault model analysis and fault types is not the main focus of this dissertation. Results from the fault analysis will, however, be taken into consideration by the design and reflected in the design phase of the ECU architecture and network topology.

2.2.3 Fault treatment

There are two main categories of fault treatment/handling concepts. The first is to exclude and contain the faulty components of a system and to prevent a total crash of the system. The second concept is to repair the faulty components, which implies fault detection, fault identification, fault isolation and fault recovery with a set-up of strategies to recover from the failed situation, perhaps in some degraded manner. A simple example of fault recovery could be the re-computation in an alternative way (indirect) of data when the related sensor fails or a change in the actuator's management strategy when one of the actuators in the system is partially or totally broken.

Fault location, to establish which parts of the system have been affected by the failure, is the basic step of both fault handling concepts. In order to localize the faults in a vehicle, they could be listed and analyzed and then classified on a probabilistic basis to select appropriate fault containment or fault recovery actions during the system requirements specification phase.

In the following two sub-chapters, the two fault handling concepts will be explained in detail.

2.2.3.1 Fault containment

As explained above, fault containment means deactivation and containment of faulty components, however the faults themselves will not be removed. Through fault containment the potential negative impact of faulty components on other components and system functionalities will be suppressed. The faulty components can be determined by means of, deactivated and replaced at the next maintenance. Fault region and fault containment region, as two of the most important terms in fault containment, are defined as follows:

- **Fault region:** A fault region is a set of components which are considered as either faultless or faulty as a whole. Within a fault region fault locations are not distinguished. If a fault region is faulty, its internal behavior is not of interest. Its external behavior, called malfunction, does no longer satisfy the specification. Depending on the “design philosophy”, fault regions range from very small to very large. Sometimes chips are taken as fault regions in the hardware. Other examples are software layers or nodes as a whole. However, in both cases the definition of fault region reflects the design strategy of a fault-tolerant system in a comprehensive way.
- **Fault containment region:** For each malfunction a maximum set of affected components can be claimed. The fault must be kept within this set, called fault containment region. Typically this requires one of the following (or a combination of them):
 - Encapsulation reduces or even avoids interaction, to make further fault propagation impossible. Reducing the interaction may affect the exchanged values and/or the timing. Generally, “fault-critical” interaction is replaced by “less-fault-critical” interaction. Typically, this refers to the specification of interfaces.
 - Fault detection makes the existence of a failure explicit. The components can then take extra actions (such as passivation) and/or perform exception handling to some extent. Alternatively, fault detection simply turns a non-detected failure into a detected one (for a larger fault containment region).

For each malfunction, the set of affected components (outside the respective fault region) where the fault can propagate to must be defined. Typically faults can only propagate via the paths of interactions. Once the fault has propagated, the affected components become erroneous themselves, and exhibit a malfunction, which must also be specified.

2.2.3.2 Fault tolerance

As stated in the above, the second category of fault handling is fault removal, where the fault can be removed or fixed during run time, so that the local crash of a function or complete crash of the system can be prevented.

There are three different methods for fault removal:

- Fault correction
- Backwards recovery
- Forwards recovery

Fault correction includes all the existing algorithms and methods, which can correct detected faults without a state change in the applications.

The backwards recovery can bring the faulty components into a consistent, former fault-free state, from which the whole system can continue working.

Compared with backwards recovery, forwards recovery brings the faulty component into a “future” state, which is designed to be fault-free.

3 State of the art: Automotive electronic architectures

After the brief introduction to dependability and fault tolerance above, the state of the art for automotive electronic architectures, including network topology, hardware, software and development process will be introduced in the following chapters, as one of the most important concepts for improving system dependability.

The past few decades have witnessed an exponential increase in the number and sophistication of electronic systems in vehicles. More than 70% of innovations (chips instead of metal) in automotive industry [McK06] are now electronic and software related. Modern vehicles carry more computation power than Apollo spaceship that flew to the moon. Many functions, such as navigation, Infotainment or engine control can not be implemented at all without extensive use of electronics. [Bro03]

In this chapter, firstly state of the art for automotive electronic architectures will be explained, where emphases are placed on the existing hardware and software architecture for automotive ECUs and the technologies, which can be used to implement highly dependable systems. After that the communication systems and development process for automotive electronics especially for safety relevant systems will be also explained.

3.1 Building blocks of automotive electronic architectures

Generally speaking there are two views when considering automotive electronic architectures:

- **Hardware architecture:** From a general point of view, hardware architecture includes the following issues:
 - ECU local structure of physical components such as processor, actuator, sensor, internal communication, bus connection and hardware methods to guarantee higher dependability, including hardware redundancy and monitoring strategy.
 - In-vehicle network topology, how ECUs are connected via the communication network in a vehicle.
 - Communication system and protocol between the ECUs.
- **Software architecture:** A description, including specification of a standard software components, a vertical interface between the layers, a horizontal interface in one layer and functionalities of the respective software components.

Before going into detail about the state of the art for automotive electronic architectures and requirements for future automotive safety systems, a few technical terms will be explained.

- **Architecture:** The general definition of architecture means the fundamental organization of a system embodied in its components, their relationships to each other and the environment, and the principles guiding its design and evolution. The term architecture here denotes descriptions of automotive electronic systems on different abstraction levels, for example abstraction from functional analysis or logical analysis.

- **Architecture Framework:** An architectural framework is an aid which can be used for developing a broad range of different architectures, which includes
 - parts of an overall system architecture into which variable components can fit
 - a method for designing an information system in terms of a set of building blocks and for showing how the building blocks fit together
 - a set of references for supporting tools
 - providing a common vocabulary
 - a list of recommended standards
 - compliant products that can be used to implement the building blocks
- **Topology:** The term topology originates from a branch of mathematics, meaning the study of that property of geometric forms. The term network topology means the study of networks in connection with non-metric geometrical properties by investigating the interconnections between branches and nodes of networks.

Topology here means the interconnection of ECUs (nodes) in a vehicle with the various communication networks and the E/E-architecture of the ECUs, including the hardware architecture and software architecture.

3.2 Requirements for future automotive safety systems

Nowadays automotive electronic systems are implemented in a distributed manner. This means that ECUs, sensors, actuator and networks from different suppliers are installed in various positions in a vehicle, forming a real-time distributed system, while OEMs are responsible for the system design and system integration/test of those ECUs. Theoretically the automotive electronic system can be also implemented in a central manner. The following are reasons for this distributed architecture:

- Traditionally, in the automobile industry a supply chain exists, in which the system supplier provides the OEM with the system components ready for direct installation.
- The distributed system enables an ideal modular oriented assembly system for reasons of packaging and cost.
- The dependability of the whole system can be improved by avoidance of "Single Point of Failure" with the help of the principles of fail-safe or fail-operational. The local fault of a sub-system can be tolerated so that the whole system should still work after the fault. The active redundant systems as in the steering and brake systems can be only implemented in a distributed manner.
- Additionally, a central architecture will introduce problems with cables and packaging, e.g. the number and length and cables will become critical if one ECU is designed to connect with all the other components.

The general requirements for automotive systems and specific requirements for future automotive safety systems will be addressed subsequently.

Real time with short latency period: A constant and pre-known latency period is the most important condition for a closed-loop control system. Here the latency period is defined as the

period of time from the time information is produced to the time that the information is consumed, where the maximum latency period will be defined in “ms”-area according to the requirements of the application.

Fault-detection: In a distributed system it is not only important to detect the local component faults, but also faults from other nodes.

Fault-tolerant: Fault-detection alone is not enough for a safety relevant system. After the fault is detected, appropriate treatment should be taken in real-time. The system should tolerate a certain, pre-defined number of faults.

Redundant-determinisms: Active redundancy requires certain determinisms, such that the internal partner should be synchronized to a global clock, so that the result can be provided in a given time window. Here, a fault-tolerant communication channel with defined latency periods is required.

Robust: In the automotive environment, electronic components are facing numerous interferences such as high, varied temperatures, vibration, mechanical impact, dust and EMC-problems (electromagnetic compatibility). Automotive electronics should be immune to these interruptions.

Composability: In the automotive industry, even if the sub-systems from suppliers are tested and validated according to the pre-defined requirement specification and every sub-system is fault free, it is still not guaranteed that the whole system will work. OEMs, as system integrator, are legally responsible for the whole system working for the customer as promised.

In order to improve composability, the interface between the system components and the responsibilities between the sub-suppliers, system suppliers and OEMs should be well defined at the design phase. Mutual dependency should be minimized, in order to improve modularity.

Testability and Certification: With the increasing complexity and multiplied model varieties, both in hardware and software, it is a challenge to test a modern automotive electronic system. In safety electronics, every possible situation should be tested, which amounts to huge efforts in time and cost. Test automation is the solution for a rapid and reproducible way to increase the depth of automotive electronic tests in a shorter development cycle.

A large percent of source codes in modern automotive software are already generated automatically, with the help of model based software engineering and a code generator. Both development process and code generator should be certificated to the appropriate safety requirements for the applications.

Maintenance and diagnosis: One of the basic requirements for automotive ECUs is the ability to diagnose for both calibration and maintenance.

3.3 State of the art: Hardware architecture for automotive electronics

3.3.1 Requirements for hardware of safety electronic systems

According to the safety norm ISO26262-5 hardware development [ISO25], the hardware safety requirements shall at least include:

- The hardware safety requirements to control external failures of the hardware item (i.e. failure occurring outside of the limits of the hardware item under consideration), with their relevant attributes (e.g. the functional behavior required for an ECU in case of an external failure, such as an open-circuit in the input of the ECU),
- The hardware safety requirements to control internal failures of the hardware item, with their relevant attributes (e.g. timing and detection abilities of a watchdog),
- The hardware safety requirements of monitoring mechanisms dedicated to indicate the internal or external failures to the driver (e.g. watchdog with driver warning),
- The hardware safety requirements to avoid and control systematic failures (e.g. safety critical timings in normal mode of operation).

As explained in Subsection 3.1, automotive electronic hardware architectures will be examined from three different points of views: in-vehicle system topology, ECU hardware architectures and communication systems between ECUs. While system topology defines a master plan of in-vehicle ECUs, the ECU hardware architecture and communication system defines the detailed hardware implementation and the communication pattern between them.

In the following subchapter, state of the art in these three areas will be explained.

3.3.2 In-vehicle system topology

An in-vehicle system topology shows the connection of ECUs in a vehicle, including:

- The number and types of ECUs
- The number and types of communication networks
- How the ECUs are connected with the communication network

In high-end vehicles, such as the Mercedes E-Class (BR211), the system topology for the communication network is highly complex, with more than 50 ECUs, as shown in the 2-D network topology in Figure 3-1. The ECUs in this case are traditionally divided into the following 4 vehicle domains; cabin/body, chassis, powertrain and telematics, where the chassis and powertrain domains are usually combined. The term vehicles domain was introduced for the connection of ECUs with a close functional relationship and connection of ECUs which are spatial closely located.

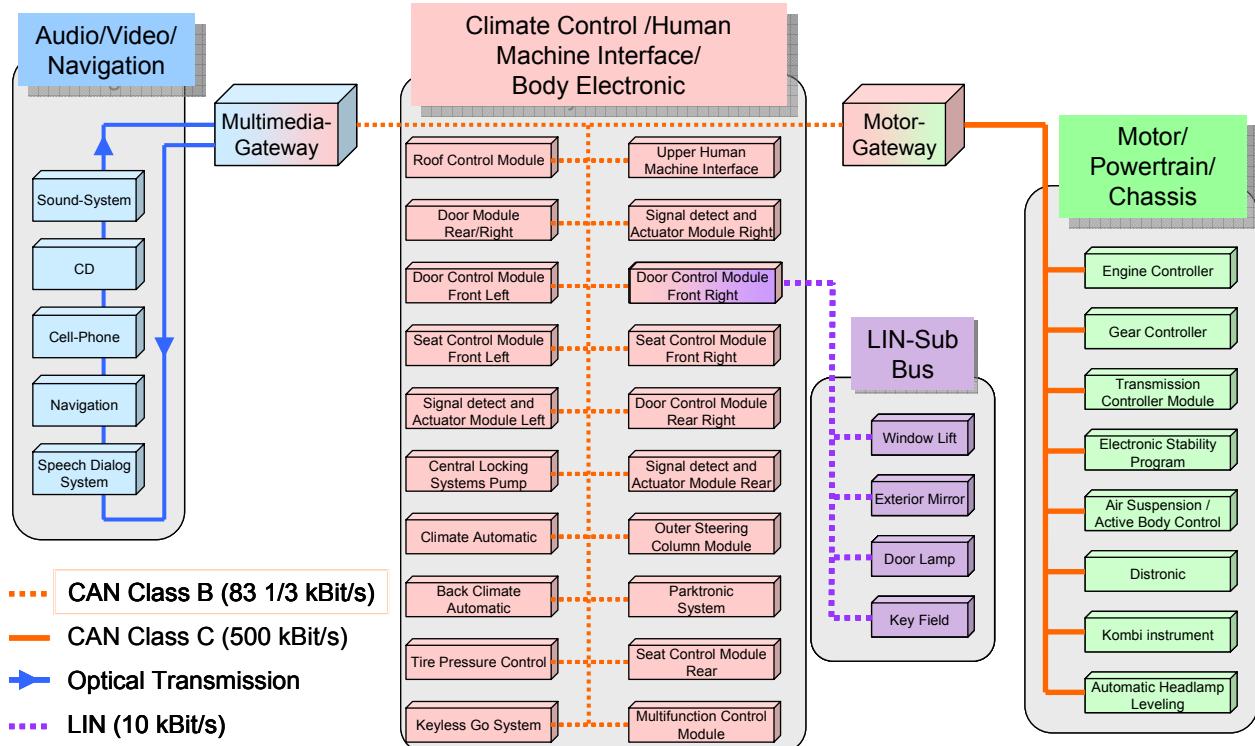


Figure 3-1: ECU topology of Mercedes E-class BR211

Modern in-vehicle network topology is primarily influenced by the concept of distributed function. Each electronic application is implemented on a single ECU as black-box. The ECUs are divided into different vehicle domains, whilst still being connected by various communication networks. With the dramatically increased number of applications in vehicles, the networking for electronic modules (sensors, actuators and ECUs) is slowly approaching the limit of mastery.

Due to reasons of cost and the requirement to reduce complexity, the current trend is towards fewer ECUs and with an increasing number of functions on each of them ([EIA04] and [ADL05]). The degree of inter-connection between ECUs will be intensified. Highly dependable applications also set new requirements for network topology, with regard to determinism and fault-tolerance.

3.3.3 ECU hardware architecture

3.3.3.1 Hardware architecture of general ECU

Generally speaking the ECU hardware can be divided into 5 main parts as shown in Figure 3-2:

- Power supply, 12 V from the in-vehicle battery
- RAM/ROM
- Interface to the communication system
- Sensors/actuators
- CPU

The other peripheries of an ECU include the analogue and digital interfaces to the sensors (A/D Converter ADC) and power semiconductor to drive the actuators and connectors.

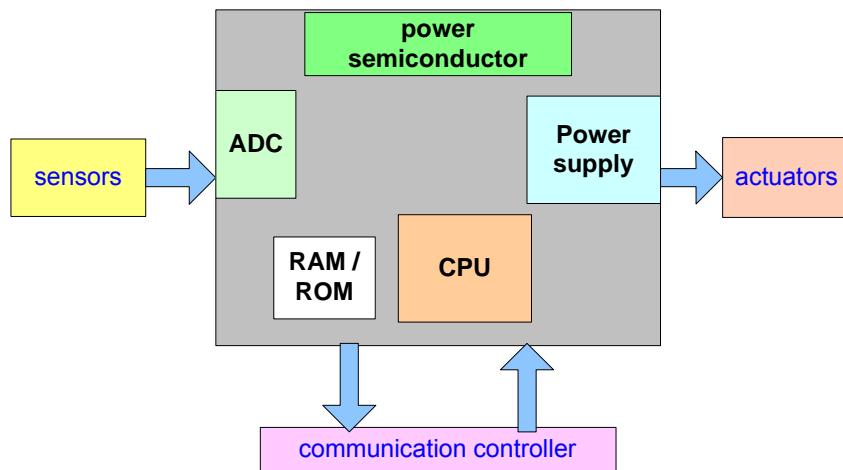


Figure 3-2: Hardware components of an ECU

Compared with domains such as chassis and powertrain, comfort and cabin electronics have fewer requirements with regards to dependability. Thus the ECUs in these domains employ the single-processor concept.

Single-Processor-ECU

The Single-Processor-ECU concept is realized with Central Processor Unit (CPU), which calculates the result of a certain algorithm and communicates with external sensors and actuators via different input and output ports. With the help of the communication driver every ECU can communicate with each other. The redundant sensors are represented from S_1 to S_n in small ellipses and redundant actuators are represented in A_1 to A_n (see Figure 3-2 und Figure 3-3).

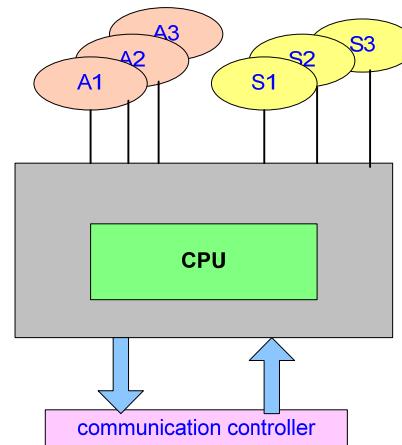


Figure 3-3: Single-Processor ECU

This kind of ECU hardware architecture, also called 1oo1 (1 out of 1 as one-channel-structure), is now most widely used in automotive electronics because of the relatively cheap material cost, considering the enormous number of deployed units. Typically, the dependability of a Single-

Processor ECU can not meet the requirements of today's active/passive safety domain and powertrain applications due to the limited fault detection capability of a single CPU, which does not have reference to a redundant unit.

3.3.3.2 Hardware architecture for safety relevant ECUs

Hardware architecture for safety relevant ECUs is characterized by high dependability, which supports fault tolerance, error detection and error handling. The traditional approach for improving hardware dependability is structural redundancy.

3.3.3.2.1 Dual-processor concept with fail-silent or fail-safe behavior

An ECU with the dual-processor concept can be implemented as fail-silent or fail-safe. The two processors work in parallel. They use the same input values, monitor each other and compare output results. Generally speaking there are two possible methods for realizing fail-silent or fail-safe ECUs, using the dual-processor concept (see Figure 3-4):

- Single processor system with a monitoring processor
- Dual-Core system with two identical processors

In single processor systems with one monitor unit, the monitoring unit, a hardware "watchdog", is usually a cheaper processor or ASIC with lower performance than the main processor. The monitoring unit runs parallel to the main processor and checks its plausibility as one channel structure with diagnosis 1oo1D-system. In case of implausibility the monitoring processor can bring the whole ECU to a safe state, for example by switching off the power supply.

Compared with single processor systems, Dual-Core systems are equipped with two identical processors with the same functionalities, which communicate via internal communication systems to compare results. The two processors communicate with an external ECU by means of a pre-defined exclusive (2oo2-system) or shared communication driver (1oo2-system). Since the same functionality can be processed on both units, a wider range of plausibility checks can be covered and therefore faults can be detected. The practical hardware realization of dual-processor-ECUs can also be different. One possibility is to realize the two processors on a common silicon chip to save costs and packaging, with the disadvantage that they can be disrupted together, while the other method is to implement the two processors in separate packages as two "independent nodes".

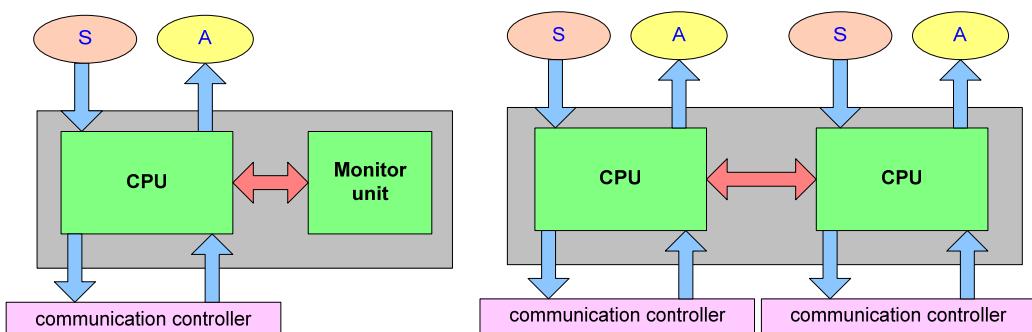


Figure 3-4: Dual-Processor ECU

The monitoring concept with a single-processor requires greater development efforts in comparison with the Dual-Core system, as two different functional entities must be implemented and tested. On the other hand, the single-processor-concept requires a processor with lower performance and thus a lower cost, meaning that production expenses can be reduced. Which of the two concepts will be chosen depends on the number of units to be produced and the safety requirements of the application. An example of the dual-processor-ECU is the modern E-Gas system or MK25 ESP-ECU [Con06] from Continental with fail-safe characteristics.

3.3.3.2.2 Fail-operational ECU (fault tolerant ECU)

The dependability realized by the dual-processor concept is fail-safe, the faulty function can return to a safe state, but can no longer be employed. For applications in X-by-Wire systems or applications in powertrain and chassis, there is no absolute safe state in many cases. Here, a fail-operational system is required, which can tolerate faults and deliver the service in spite of them. The general approach for progressing from a fail-safe system to a fail-operational system is to raise the number of redundant processor units by means of simultaneous, mutual monitoring. There is currently no real fault-tolerant automotive ECU in practice or in serial production. However, from other domains, for example in the aerospace industry, and related literatures there are a number of hardware topologies, which can realize a fault-operational ECU.

For the automotive industry only two of these will be considered, for reasons of complexity and cost:

- Triple-processor (triplex) ECU
- Dual-duplex processor ECU

Triple-Processor (triplex) ECU

The Triple-Processor ECU works in the same way as a dual-processor ECU, but with 3 processing units performing mutual comparison of data and state during data processing, instead of 2. Usually, every processing unit has its own communication controller (2oo3-system) as shown in Figure 3-5.

In order to reduce production costs, only 2 out of the 3 processing units are normally connected to the sensors, actuators and external communication system. The third microprocessor receives the data via the internal communication system from the other two processors and in case of difference during comparison an agreement protocol 2 out 3 will be performed, as shown in Figure 3-6.

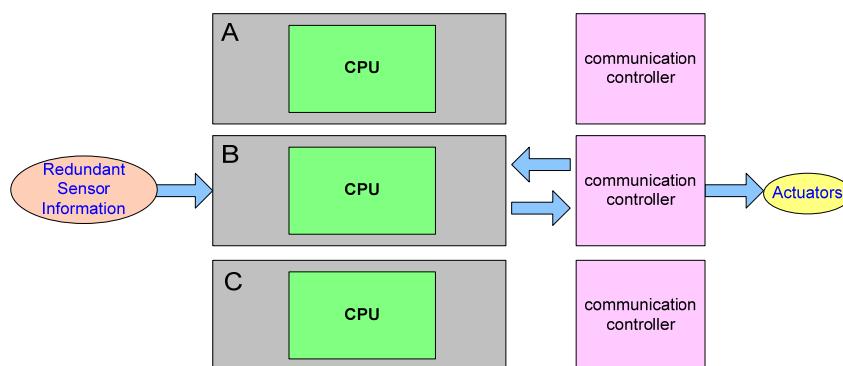
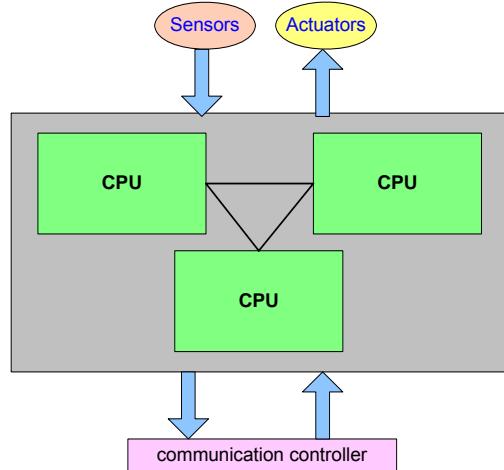


Figure 3-5: Triple-Processor ECU as a full 2oo3-system

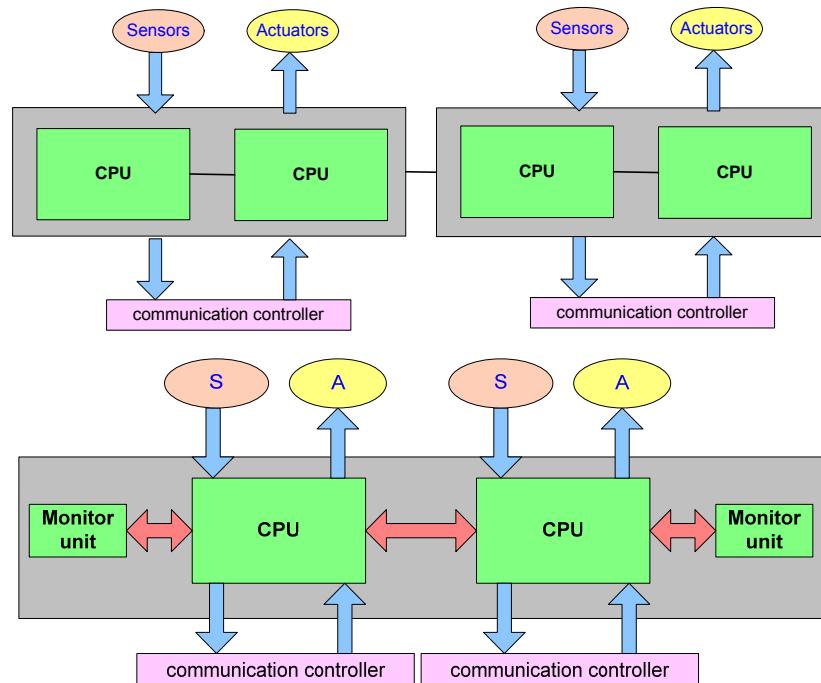
**Figure 3-6: Triple-Processor ECU**

In the Triple-Processor ECU, even when one processor fails to work, the other two processors can still perform fail-safe behavior, because they can still monitor each other and shut off the system in case of inconsistency (under the situation that I/O and communication channels can be redundantly controlled). The disadvantage of such a system is the greater effort required to compare the input value, results of the processor, state, etc.

In order to simplify the agreement process between the processors, a hierarchical structure, similar to the following system, can be employed.

Dual-Duplex-Processor ECU

Such hierarchical structure can be realized in a Dual-Duplex-Processor ECU. Initially, it appears to be a simple duplicated dual-processor ECU with internal communication between them, which informs the other about the state and data to compare. As shown in Figure 3-7, this initial variant is fully symmetric. By duplication of single processor with a monitoring unit, the dual-duplex architecture can be also built up as the second variant in Figure 3-7.

**Figure 3-7: Two variant of Dual-Duplex-Processor ECU**

3.3.3.2.3 Power supply for safety relevant ECUs

While dealing with safety critical functions and redundant fail-operational systems, it is important to avoid any “Common Cause” of failures. One such typical failure cause is a power supply failure. Although the topic of power supply management is not the focus of the work here, a few important points will be explained.

A fault tolerant power supply, which is essential for fail-operational systems, also requires redundancy within the power supply. Redundant power supply lines must be mutually insulated, in such a way that a failure on one must not propagate to another. Power supply strategies shall ensure fault isolation between the power supply of critical and non-critical parts in order to avoid faulty propagation. The following characteristics of the power supply system are recommended for safety relevant systems:

Power sources disconnection: disconnection of the power sources (storage units and power generation system) must be foreseen.

EMS (Energy Management System): maintains and monitors Electrical Energy Balance and ensures a proper state of charge to the storage systems (batteries)

BMS (Battery Monitoring System): accurate estimation (measurement) of the status of power storage systems.

Fuse Boxes: proper protection shall be carefully studied for each load, with particular regard to the fault isolation principle.

EUS (Energy Uncoupling System): is a component used to uncouple the Power Generator supply bus and the redundant supply buses under certain failure conditions.

3.3.3.3 Conclusion for ECU hardware architecture

State of the art in automotive hardware architecture for non-safety critical application currently remains single-processor unit with simple hardware watchdog and directly coupled actuators and sensors. With the increasing requirements of highly dependable systems, especially for future X-by-Wire applications, appreciate hardware architecture with redundant backup units is applied to increase the fault-tolerance of the system. Existing concepts of fail-operational ECUs, however, still need to be standardized and adapted for automotive applications.

3.3.4 Automotive communication systems

As shown in Figure 3-1, four different communication networks are used in the Mercedes E-Class. However, such heterorganic communication networks are no longer simply the privilege of luxury vehicles; modern, mid-range vehicles also employ several different communication networks.

3.3.4.1 Requirements for automotive communication systems

Before the various communication networks employed in modern vehicles are considered, it is necessary to consider the requirements for automotive communication systems.

- For ECU communication, configurable synchronous and asynchronous data transfers are both required.
- Many applications, especially safety relevant applications with control loops, require a deterministic manner of data transfer with QoS.

QoS here means to the ability of a network element, such as an application, network node or gateway component, to have some level of assurance that its traffic and service requirements can be satisfied. Enabling QoS requires the cooperation of all network layers from top-to-bottom, as well as every network element from end-to-end. There are four key QoS parameters:

- Latency (end-to-end time for a service, until the time service is delivered to the application, including propagation delay through network as one part)
- Jitter (delay variation)
- Bandwidth
- Packet Loss

In time-triggered communication networks such as FlexRay, QoS is realized by static communication scheduling. In event-triggered communication systems such as CAN, important signals are sent periodically, but in particular non-periodic diagnosis messages, are sent dynamically. In telematics applications, when concerning mobile communication networks or IP-based networks, QoS is realized dynamically.

- Fault tolerance in communication: for the QoS, some kind of fault tolerant communication is also needed, to ensure the data integrity when one bus fails to transfer data due to certain physical failures.
- Scalable support for redundancy: redundant units such as actuators and sensors must also be coupled with ECUs in the network. Moreover, when redundant processor units are realized by separate packages, as independent nodes, the communication between the redundant units is also provided by the communication network.
- Global time base: a global time base is of vital importance for a time-triggered system, where the local time in each ECU is synchronized with the global time.
- Most safety relevant applications require deterministic behavior, including data transfer. The state of the art for in-vehicle communication by means of CAN-bus is based on

arbitration, in which messages with different priorities compete with each other for bus access. Each node can access the bus coincidentally, thus no peak data transfer rate can be foreseen at the development phase. This means a deterministic data transfer with deterministic bus load can not be guaranteed.

- The bus load in state of the art in-vehicle communication varies from a few Kbit/s to a few hundred Kbit/s. The transfer rate, especially in the domains of powertrain and cabin, is relative low. However, with the increasing complexity of applications and data fusion, the future demand for a higher data transfer rate of up to 10 Mbit/s is a possibility.
- Compared with the communication networks in IT, in-vehicle bus communication has a number of constraints to consider, including isolation, EMC (electromagnetic compatibility) and assembly. Therefore, the communication network should support both optical and electrical transfer mediums.
- The in-vehicle communication system should also provide support for an energy management concept for a long run battery period.

3.3.4.2 State of the art in automotive communication systems

In this section, different in-vehicle communication busses, such as, CAN, ASRB, Byteflight and FlexRay will be briefly introduced. Other in-vehicle communication networks such as LIN (Local Interconnect Network), TTCAN (Time-triggered CAN [Har02]), MOST, IEEE-1394 are not the current focus of the work and will be disregarded at this stage.

- The most widely applied automotive communication network is Controller Area Network (CAN, [ISO11898] Part 1-3). CAN is an event-triggered communication network with a transfer rate from approximately 20 Kbit/s to a maximum of 1 Mbit/s. Bus access for CAN follows the arbitration concept based on the priority of each CAN-message. This implies that when more than one node wants to access the bus medium simultaneously, only the message with the highest priority can be sent. The messages with lower priority will be withdrawn and sent at a later time, thus the time period to access the bus is not deterministic, since a message with low priority can be withdrawn many times without success and other CAN-messages could dominate the bus.
- Automotive Safety Restraints Bus (ASRB) (former Safe-by-Wire) [ASR04] is a sensor bus for the automotive passenger restraint systems. The automotive passenger restraint systems are controlled by a "Restraint Control Module" (RCM), which is connected to peripheral devices.
- First published as "SI-bus" in 1999, the Byteflight bus [Gri00] was developed to connect passive safety systems, such as airbag ECU and crash sensors. As a communication schema, Byteflight uses the FTDMA – Flexible Time Division Multiple Access algorithm. Compared with modern in-vehicle communication systems it provides a much higher bandwidth owing to the fast physical layer (10 Mbit/s) and a star-topology. The total intelligence is based on the so-called Star Net Coupler, which synchronizes the bus and controls bus access. Messages in Byteflight are sent cyclically in Mini-Slotting algorithms. The state of the art in Byteflight consists of plastic optical fiber, which is not suitable for current automotive application (heat-resistance, torsional radius, etc).

- The development of FlexRay began in 1999 shortly after the development of Byteflight and aimed to eliminate the disadvantages of Byteflight. FlexRay, ([Bel01] and [FxR02]) is the state of the art for fault tolerant communication system. It fulfills the requirements for safety relevant applications and future X-by-Wire systems. FlexRay provides two configurable transfer modes, synchronous and asynchronous. Cyclical messages with real time requirements can be transferred in the synchronous mode using a TDMA algorithm, while event triggered messages or diagnostic messages can be transferred in the asynchronous mode using an FTDMA algorithm, similar to Byteflight.

Due to the flexibility between the dynamic and static segments, FlexRay can be used as purely time-triggered or mixed with an event-triggered communication system, as shown in Figure 3-8.

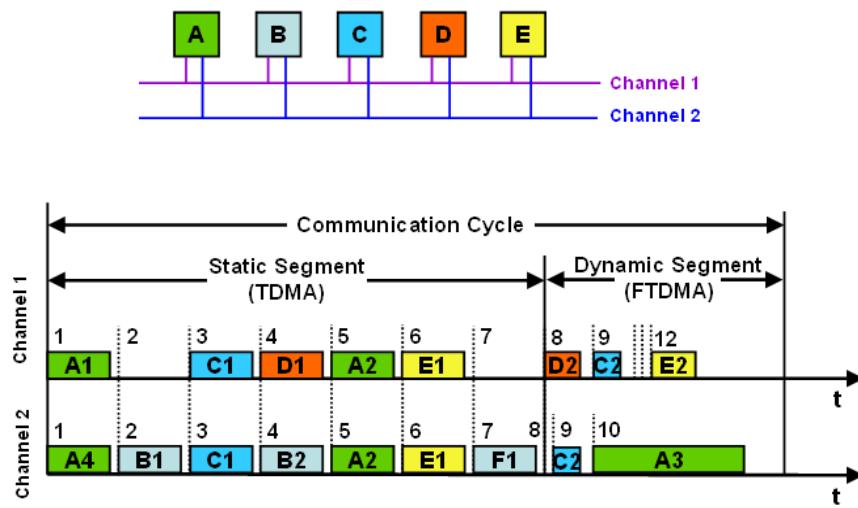


Figure 3-8: FlexRay bus access and communication cycle

One of the most important new features of FlexRay is fault tolerance, not only with the fault tolerant mechanism from the physical layer, for example CRC etc., but also directly from the protocol layer, which enables a scalable redundant system with higher dependability. A cycle counter is implemented in FlexRay which monitors the sequence of messages sent and received. Moreover, end-to-end communication safeguarding and timeout monitoring can also be implemented as extensions to the software.

FlexRay supports different network topologies with the following basic topologies:

- Point-to-point
- Passive bus
- Passive star
- Active star (capable of deactivation of faulty network branch or nodes [Ele04])

More complex topologies as depicted in Figure 3-9 are possible by combining the basic topologies as follows

- Cascading of active stars
- Combining bus- and star-topologies to a hybrid topology
- Combining dual FlexRay channel facilities with the above topologies to: dual busses, dual stars and dual hybrids

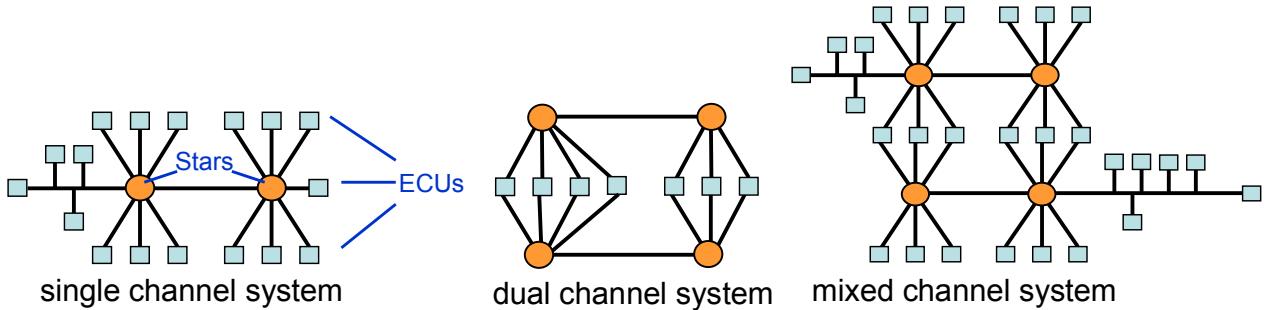


Figure 3-9: FlexRay topologies

The star-structure, compared with the passive star (ground connection as in CAN) is an active node. Physical failures, such as short-circuit, will be detected by the active star node and recovered from by turning off the damaged branches. Failures in the time domain e.g. by nodes jamming the bus as a “babbling idiot” FlexRay can prevent by the optional local or central “Bus Guardian”.

The other characteristic of FlexRay (compared with Byteflight) are listed below:

- Redundant channel including bus monitoring
- Distributed time synchronization
- No intelligent Start Coupler as single point of failure
- Maximum of 2 star structures can form a daisy chaining
- Electronic physical layer with transfer rate of max. 10 Mbit/s

3.3.4.3 Conclusion for automotive communication systems

Current in-vehicle communication systems exhibit a high diversity in sub-networks, regarding hardware, transfer velocities and communication protocols. CAN, as the most common in-vehicle communication network, has established its place as a mature technology. In order to gain design correctness and deterministic communication with a higher bandwidth, to master the complexity at system integration, new communication networks, such as Byteflight and FlexRay have been implemented. For future integrated safety applications, FlexRay is considered the most suitable candidate for realizing fault-tolerant communications between distributed safety functions.

3.4 State of the art: Software architecture for automotive electronics

Besides the hardware architecture, the software architecture is one of the most important parts of an automotive electronic system. Currently, a large number of ECU functionalities in vehicles are implemented in software [McK06]; consequently according to a “post-modern” reliability theory [Kop02], software has become a serious problem for the dependability of embedded system.

3.4.1 Requirements for software in safety electronic systems

The following lists some of the general requirements for software in automotive safety electronic systems [SWR07]:

- Modern automotive software systems are highly complex and dynamic, for example every engine ECU can have a specification with more than 3,100 pages, 7,500 variables, 4,900 parameters and 500 system constants. The management of different versions of software modules on different ECU hardware alone has been a problem.
- The applications implemented in current software are mostly distributed systems with functionalities, which are implemented over different domains, making use of the synergy-effect for flow of information over domain borders.
- Compared with normal distributed systems, automotive software requires high real-time in latency, jitter and other QoS parameters. Many safety relevant applications are time triggered, thus requiring deterministic behavior both in the OS and SW-Cs, for example deterministic execution time.
- Many error detection mechanisms are now implemented in software, e.g. CRC-check. Different methods for the distribution of information concerning fault detection in real-time, for example atomic, broadcasted and asymmetric, can be employed.
- Fault-masking in redundant systems requires determinism in software, the results of which can be synchronized at the output ports of the processor units to make a unified decision with the voting-protocol.
- Resource management: Compared with the IT-industry, the embedded system has limited resources in terms of processor performance (run time of processor unit), memory (RAM/ROM), and peripheries. Services for resource management should ensure that higher level safety integrity systems always dispose of resources when resources are shared with different units, across integrity levels.
- Individual monitoring of software components: To implement fault tolerance mechanisms for safety relevant applications integrated on the same HW-platform, individual monitoring mechanisms are necessary. They should be available in an OS or implemented by other means, for example by checking whether the timing of a SW-C is within expected ranges or whether all required SW-Cs for an application started correctly.

- Reliability and robustness of automotive software is another requirement for ECU software, so that fault tolerance mechanisms can handle detected faults locally without propagation to other SW-Components, for example mechanisms for regional fault containment to avoid safe functions being contaminated by faulty functions.
- Composability of automotive software is of vital importance for system integration by OEM and system suppliers. It must show a certain level of flexibility for the configuration, allowing the user (developer) to specify certain fault-tolerance characteristics and provide the requested properties. Different safety relevant applications (and even their sub-systems) may have different safety integrity. It is important to partition the applications to manage the complexity by the integration.
- The test and certification of automotive software according to the state of the art safety norms is also an important issue, especially as the development time for automotive software becomes ever shorter.
- Maintenance and diagnosis is an important part of the software life cycle, especially for integrated applications involving more ECUs and fault-operational behavior. New diagnosis mechanisms should be researched.

3.4.2 ECU software architecture

ECU software can be divided into two general categories:

- application software
- runtime environment

The application software includes parts of the software, which vehicle users (drivers and passengers) can directly feel (for example ABS or seat adjustment).

Compared with the application software, the runtime environment is everything else in ECU software, which provides services required for the application software (for example operating system and basic software modules). The main task of a runtime environment is the abstraction of underlying hardware, so that the application can be implemented more or less independently of the ECU hardware details. This ensures that the application software can be easily transported and integrated into another ECU.

In the following, a few main concepts from the trends of development in ECU software architecture will be explained.

3.4.2.1 Concept of standard software components

The state of the art in automotive software is the standard software components (Standard Core). The software components are modular and the runtime environment is configured statically to the time of development. The same standard software modules are used in all ECUs by an OEM.

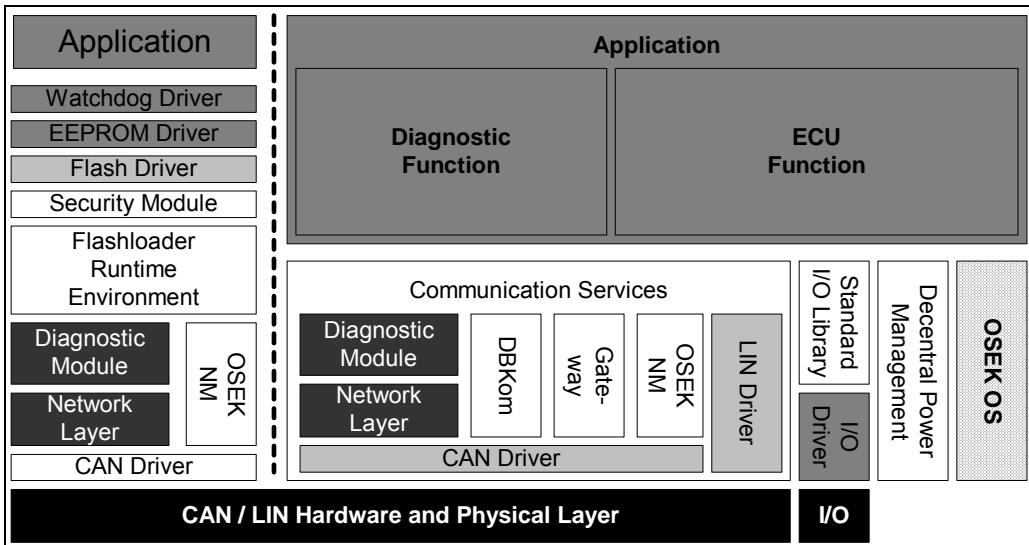


Figure 3-10: Software topology for Mercedes-Benz cabin ECU [Ruh04]

Figure 3-10 shows an example of an automotive software topology, which can be divided into two parts. The part to the left of the dashed line is for flashing; to configure and load the functional software onto the ECU by means of maintenance. The part to the right includes the application software modules and runtime environment, on top of which is the application software with ECU and diagnostic functions. Within these applications are the statically configured standard software modules, which communicate with the hardware through the defined APIs, including the following functionalities:

- Communication
- Diagnosis
- Parameterization of software
- Gateway
- Network management

3.4.2.2 Concepts of software platforms

Another possible concept for the software architecture in a distributed system is the “middleware”-concept or software platform concept. Compared with the standard software concept, middleware provides further abstraction for the underlying communication hardware and processor unit.

In the IT-industry, the current most popular concept in middleware is the “Common Object Request Broker Architecture” (CORBA). All communication mechanisms in CORBA are based on the client-server request-response concept.

A special version of middleware, modified for automotive use, was implemented in the project TITUS [Ruh04]. Here, an extra software layer for the communication was defined. The client-server concept can fulfill the requirements for event-triggered communication in a vehicle quite well. However, many functions in the control loop require a cyclical signal and a highly dependable data transfer considering time, in this case the client-server communication is less suitable.

Concept from EAST-EEA project

The EAST project (Electronics Architecture and Software Technologies) took place between mid 2001 and mid 2004, with the aim of developing a middleware based on modern standard software modules (www.east-eea.net). It defined a standardized open software architecture framework, as shown in Figure 3-11, which enables hardware-independent portability of software modules. At the very bottom of this architecture framework, there is micro-controller with the attributes of CPU power and memory size. Above it, Embedded Basic Software includes components such as operating system, hardware drivers, communication functionality system, diagnosis, hardware drivers and hardware abstraction layer. The Embedded Basic Software can not be seen directly by customers, so it is not relevant for competition in the market. With the help of EAST-middleware, different applications can be mapped onto one ECU.

This middleware offers services for transparent communication inside the classical vehicle domains as well as for cross-domain communication. Subsequently, certain selected domain oriented approaches for the body and powertrain domains and the concept for the implementation of cross-domain services are described in the EAST-project.

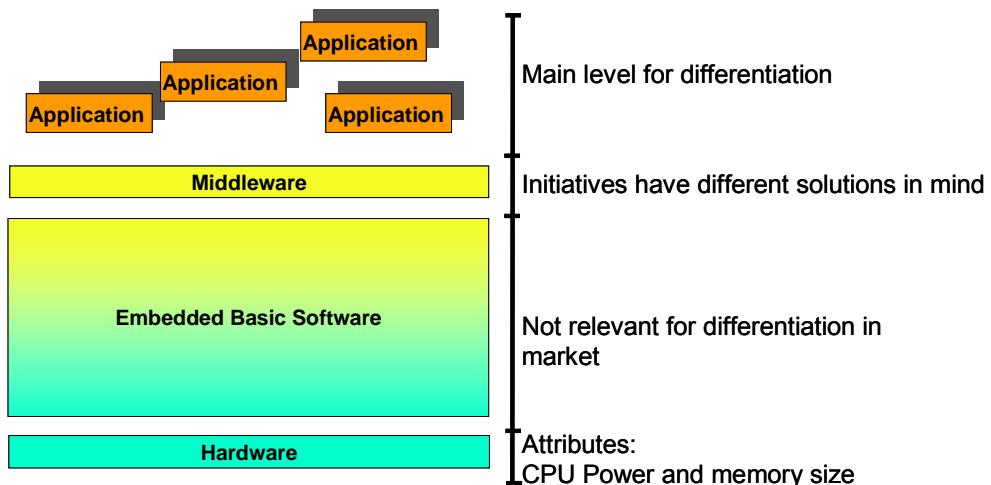


Figure 3-11: EAST architecture framework

Concept of AUTOSAR software platform

The AUTOSAR project (Automotive Open System Architecture, www.autosar.org) [AUT07], [ATO07] is a an alliance of OEMs and tier-1 automotive suppliers, who aim to develop and establish an open industry standard for an automotive E/E architecture, which will serve as the basic infrastructure for the management of functions within both future applications and standard software modules.

A general ECU software architecture, defined in AUTOSAR, is shown in Figure 3-12, which is similar in many features to the architecture framework defined in EAST.

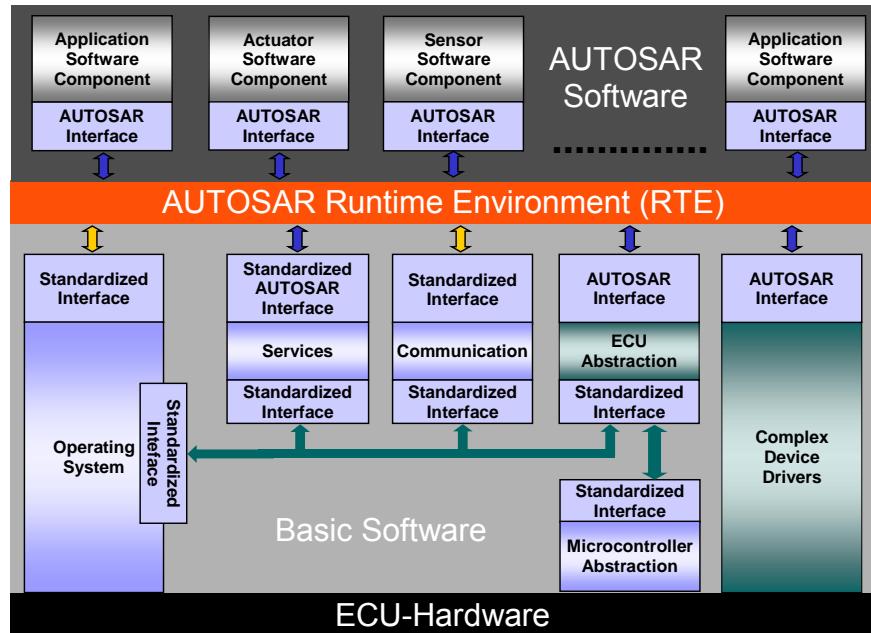


Figure 3-12: AUTOSAR architecture framework [SWA06]

In the AUTOSAR software architecture, SW-Cs of automotive functionality will be encapsulated by the AUTOSAR interface, which ensures the connectivity of software elements surrounding the AUTOSAR runtime environment.

AUTOSAR interfaces should not be considered as pieces of software dedicated to adapting any software module to an AUTOSAR environment. Rather, they are an integral part of each software module running in an AUTOSAR compliant electronic control unit.

In the place of EAST-middleware, an AUTOSAR-middleware called RTE (Runtime Environment) is defined.

- At system design level the AUTOSAR RTE acts as a communication center for inter and intra electronic control unit information exchange between architectural elements of layers connected to the AUTOSAR RTE. All communication between AUTOSAR components has to be compliant with the standardized AUTOSAR interface definitions. All other architectural elements have to provide standardized interfaces that are primarily based on standards which already exist.
- At implementation level the AUTOSAR RTE will be at least reduced to those communication channels which establish communication between SW-components allocated on different electronic control units.
- Different electronic control units will most likely run different AUTOSAR RTE. The AUTOSAR RTE will therefore be assembled with the help of a generator tool by using standard building blocks.

The software layer underlying AUTOSAR RTE is called “Basic Software”. It is a standardized software layer, which provides services to the software components and is necessary for the running of the functional part of the software. It does not fulfill any functional job itself and is situated below the AUTOSAR Runtime Environment. The basic software could contain

- NVRAM management
- Transport layers for different communication technologies (e.g. CAN, LIN, etc.)
- Network management
- System services such as diagnostic protocols, etc.

Here it is to note that AUTOSAR partnership defined not only the software architecture. In AUTOSAR, UML2.0, certain standardized notation, methodologies and guidelines were applied for the design specification, where a significant effort is invested in the harmonization and unifying the languages between European OEMs and suppliers. Hence, not only the software architecture but also the developing language can be standardized and harmonized. The efficiency in the cooperation between OEM and suppliers can be improved, e.g. the tracing between requirement specification and design specification can be eased and misunderstanding can be reduced.

3.4.3 Automotive software services

Modern automotive software can already provide a large variety of services. In the following a short overview of the software services is given:

- Communication services: Asynchronous and synchronous communication services between applications running on the same or different ECUs.
- Resource management: The current ECU operating system OSEK OS 2.1 [OSK05] can already provide some resource management considering the processor running time. All tasks are assigned with a certain priority, statically in the development phase. OSEK, as a pre-emptive real-time multitasking operating system can arrange the tasks so that the tasks with higher priorities can receive processor time, without being blocked by the lower tasks.
- Network Management for energy: With the introduction of more and more ECUs with ever higher performance, energy consumption has not only been a problem for high end vehicles. Software services for network management ensure a synchronous start-up and sleep of all ECUs, thus the life cycle of the battery can be longer.
- Function management: The OS also provides mechanisms for terminating functions in other processes, in case of timing violation or demand from tasks with higher priorities.
- Memory management: Some basic access managements are provided by current automotive software, where unauthenticated access of read/write in RAM/ROM is prevented.
- Event Notification: with the flag – set und reset certain bits of state vectors. Task activation and call-back-routine can be called in the operating system.
- Diagnosis of applications and automotive middleware: With defined error symptoms, some run-time errors can be detected by the OS and an alarm will be sent. Further support for fault-tolerant systems, however, is not available.

- Calibration services for automotive software: By means of assembling/flashing or maintenance of ECUs, the software will be calibrated.
- Gateway services: As in the IT-industry, it is sometimes necessary to transport messages and signals from one domain to another. Routing, modification and rearrangement of signals are supported by current ECU software.
- Hardware abstraction: To improve transparency for sensors, actuators and micro-processors in ECUs, hardware abstraction of a certain degree is provided by state of the art ECU software.

3.4.4 Conclusion for automotive software

Automotive ECU software architecture exhibits a clear trend to the structure and abstraction of hardware, where software will be implemented for the distribution of the functionalities and the available communication drivers. This means that the mapping of ECU application software will be more flexible. Compared with standardization and high level abstraction in the IT-industry, which enables an unproblematic distribution and integration of software on any IBM compatible PC, automotive software architecture still has more potential for standardization. For safety relevant applications, especially for applications involving several ECUs across domain borders, there is still no ECU software with fault-tolerant middleware, dependability software services and deterministic behavior in automotive systems, which are all necessary for the integrated safety concept.

The project EAST-EEA, which has already been completed and ongoing projects, such as AUTOSAR, both demonstrate a clear effort on the part of automobile OEMs and suppliers towards a common automotive electronic architecture with emphasis on a middleware concept. Through the standardization of software applications and basic software services, the automotive software quality and test depth can be improved. As state of the art, there are also concepts and prototypes about dependability software services like FT-COM, OSEK-Time, HIS specification about Protected OS, etc. But they are all more or less isolated solutions without an architecture framework.

AUTOSAR, as a significant contribution towards in-vehicle embedded software platform, has the aim to design an architecture framework for all the in-vehicle domains. The original aim of AUTOSAR was to provide a common ECU electronic architecture framework for all automotive domains. However, safety domain is not the topic of AUTOSAR phase 1. The safety relevant applications implemented on ECUs in domains such as chassis and powertrain have their own specific requirements for system dependability and fault-tolerance both in data communication, software and hardware, redundancy management and diagnosis for applications spreading over a few domains, etc.

For Integrated Safety Systems, a special architecture framework with hardware, software, communication systems and a development process, taking into consideration the results of EAST-EEA, HIS and compatibility with common automotive architectures, in particular AUTOSAR, need to be defined.

3.5 State of the art: Development process for automotive electronic systems

In the above chapters, the automotive electronic architecture has been discussed, with emphasis on safety systems mainly from the perspective of hardware and software architectures. Another important topic in the architecture framework is the development process for automotive electronic systems, because electronic architectures can not be separated from the development process. It is application that drives new functions and functions create new requirements for the architecture and innovations in the architecture. The new architectures then drive changes in the development process.

3.5.1 Requirements for the development process of safety electronic systems

Before the technical details of different existing development processes and concepts are explained, a special look at the requirements for the development of electronic safety systems is meaningful.

- For safety relevant electronic systems, in the system analysis phase, safety specification and risk management for safety systems should be carried out. A Safety Requirements Specification (SRS), including a systematic, disciplined and quantifiable process for hazard analysis should be initiated in the engineering phase, for example hazard analysis and risk assessment including engineering risk management aspects of the vehicle, driver, and the environment, can be included in the engineering phase.
- Compared to the hardware development, in which the required safety integrity can be ensured by analyzing the probability of failure, the safety integrity involved in software development is reflected exclusively in higher requirements made on the development processes.
- With the increasing complexity of automotive electronic systems, the complexity of requirement specifications is also increasing explosively. The version and modification management of requirement specifications is a challenging theme. After a supplier provides an OEM with the developed product, the electronic systems will be tested against its specification. How to guarantee a traceable specification for system architecture and already take the tests into consideration in the specification phase is an important topic.
- In the design phase and system implementation phase, another problem is how to correctly transform specifications into design and design into source code.
- Early validation and verification is of vital importance for the development of safety relevant systems to improve the quality and shorten the developing time.

3.5.2 State of the art in development processes

In the state of the art development process there are a number of models, which can be used to structure and define the development stage. In the following, the most general development process, the V-model, will be briefly introduced. The general development processes will be tailored and modified for the development of automotive electronic systems with regard to the special requirements for automotive safety electronic systems.

3.5.2.1 V-Model

The development model, V-Model, is almost standard in the automotive industry. The principle of the V-Model is the description of the relationship between development and test steps, as shown in Figure 3-13. Every development step is referenced to the test step, which requires a lot effort (almost 40%) in the last phase of development. This implies a disadvantage because design failure can be only detected in the later phase of the development, which requires more time and expense.

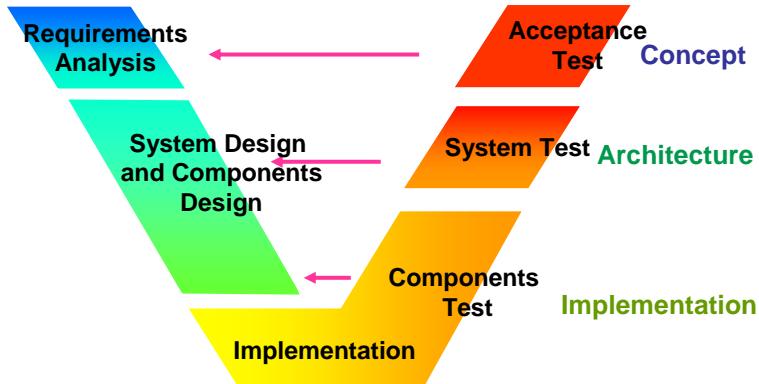


Figure 3-13: The V-Model

3.5.2.2 Development processes in automotive electronic systems

Based on the general engineering process, a specific development process for automotive electronic systems is generated and adapted for the requirements of the automobile industry.

3.5.2.2.1 Model and simulation based development process

The model based development process (sometimes also called simulation based development process) is realized on the basis of the Spiral Model, in which a sub-system will be developed iteratively. It will be tested in a pure software environment with the help of simulation tools and the functionalities of the application to be developed (the “software in the loop” test). The implementation of software on a particular hardware platform will only be initiated after a successful test of the functionalities with the help of models. In many cases, a code-generator will be used in the implementation of models, as shown in Figure 3-14 [Ruh04].

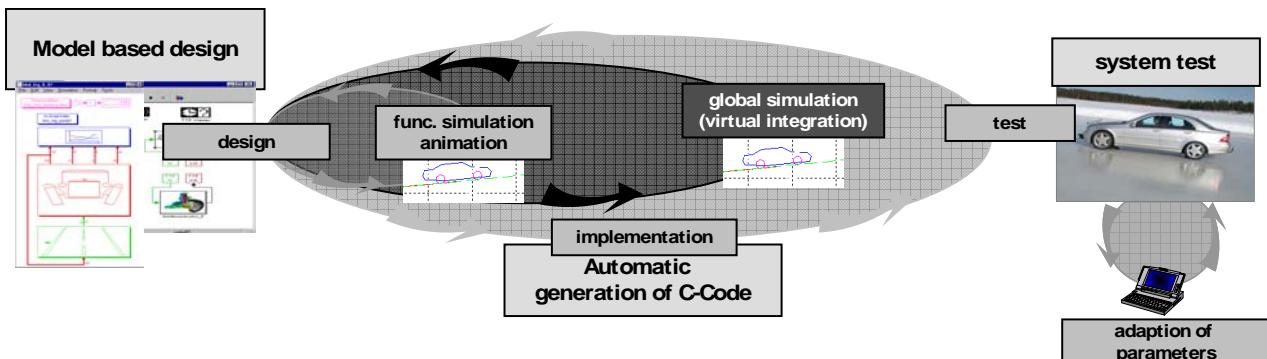


Figure 3-14: Model based development process [Ruh04]

Different development environments/tool chains for the model based development process currently exist. The most widely used simulation tool, especially in automotive electronic systems, is Matlab/Simulink. All these tools or tool chains have one common aspect; the functions/application will be verified using a simulation tool, with less consideration for the underlying hardware. In this way the conception decisions, for example mapping of functions onto the ECU and topology in the later development phase, can be evaluated in the early phase

3.5.2.2.2 Phase oriented development process

The automotive industry currently develops products in several design phases that are mainly driven by the requirements of the mechanical engineering process. This overall design process is often called the lifecycle engineering process. This lifecycle engineering process includes different management milestones (for example engineering start, styling/package freeze, release and start of production). During this process, the vehicle manufacturer produces simulation models and development car versions for different purpose (mules, varying versions of prototypes). The goal of these vehicles is to test the different components of the car. This testing includes (depending on the prototype version) component development, vehicle integration, and performance evaluation. Traditionally, the testing process steps are based on the requirements for the mechanical parts (for example engine, chassis); however, the testing of automotive electronics requires more and more attention.

To allow for vehicle development at the OEM with the most adequate electronic parts, the development process at the supplier has to be adapted to the manufacturer's process. Therefore, the building blocks according to the V model are performed not only once, but several times. The hardware result of each pass of the V cycle is called a sample (for example A-Sample, AB-Sample, B-Sample, C-Sample and D-Sample as depicted in Figure 3-15). These samples are implemented in different development stages of vehicle manufacturing. A-Samples here are defined as functional models with limited suitability for driving with constraints. There is no requirement demanding the existence of safety functions. B-Samples are samples with suitability for driving guaranteeing sufficient operational safety for first trials in the test bed and in the vehicle. The reliability of parts is dimensioned for driving operation on a test area. Safety functions are completely implemented. The C-Samples are ECUs built by production tools with constraints close-to-production. All requirements for function, reliability and interference resistance shall be complied with. The reliability of parts is dimensioned for driving operation on the road. The D-Samples are ECUs built by production tools with production constraints. The electronic control units are completely applicable.

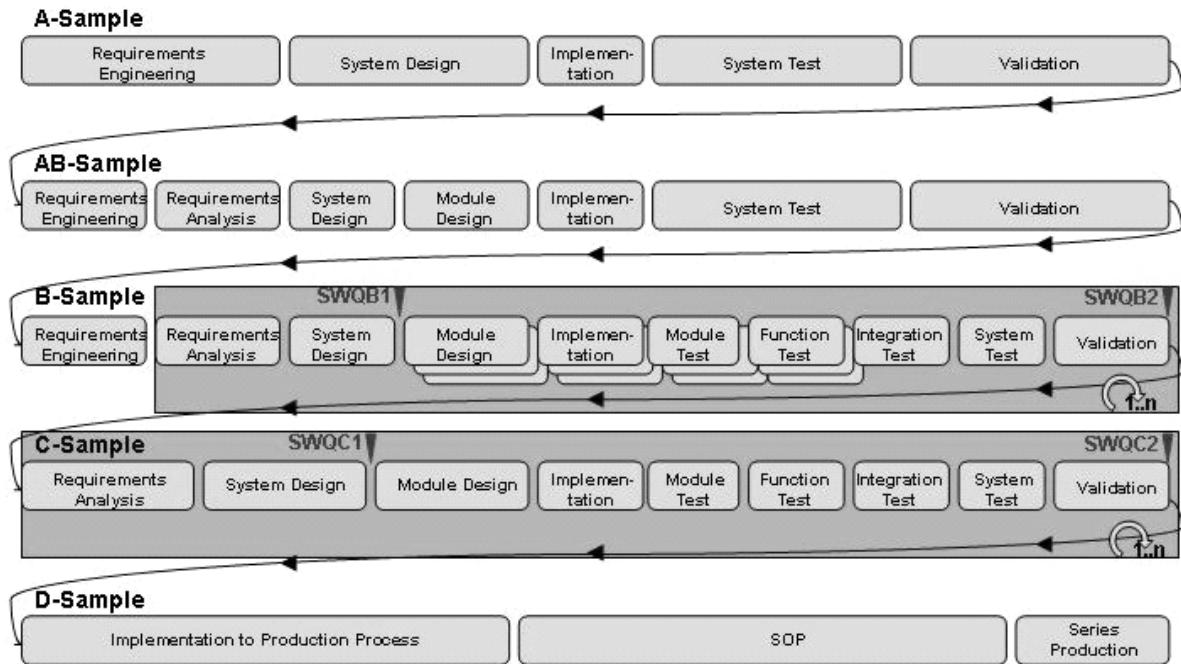


Figure 3-15: Basic V-style process model integrated in lifecycle engineering process

3.5.2.2.3 The development process for fault tolerant electronics according to Hedenetz

For the development of safety relevant systems, Dr. Hedenetz [Hed01] has modified the process described above and adapted it to safety relevant systems (as shown in Figure 3-16).

Hedenetz divided the development phase into global design and local design. In global design the whole system is considered, while in local design the details and relationships between the OEMs and suppliers are considered. In the development process of Hedenetz, simulation process is the coupled as a central part of the development, where the constraints of high dependable systems are taken into consideration.

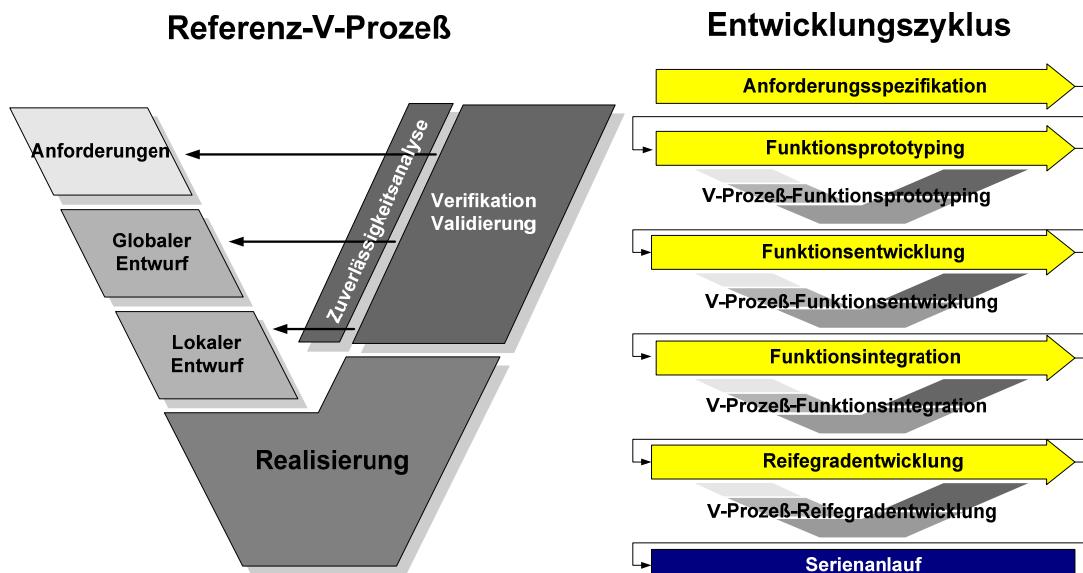


Figure 3-16: Development process according to [Hed01]

3.5.2.2.4 The development process for safety electronics according to Benz

Benz [Ben04] introduced a “Double-V-Model” for the development of automotive safety relevant electronic systems, as shown in Figure 3-17. With the help of two separate but inter-connected V-models, one for the traditional V-model, the other for the safety, the additional requirements in quality and system dependability can be fulfilled. The system safety in the development process must be given more attention compared with the existing process. Safety is a characteristic of the whole system and should be considered in each component, thus both V-Processes should be of equal importance as “functional safety co-design”, while the term “functional safety” is defined as following: Safety that a vehicle function does not cause any intolerable endangering states resulting from specification-, implementation or realization errors and failures while operation period reasonably foreseeable operational/users errors and reasonably foreseeable misuse.

In the Safety-V, parallel to the functional design, a functional hazard analysis will be performed, the potential impact of the safety function on its environment, including vehicle, driver and other traffic users will be analyzed and an appreciate safety level given. At the system design stage, with the help of the system safety assessment, different system architectures will be evaluated from the point of view of safely integrity. The implementation and verification/validation phase will be also evaluated as proof of fulfillment of the safety requirements.

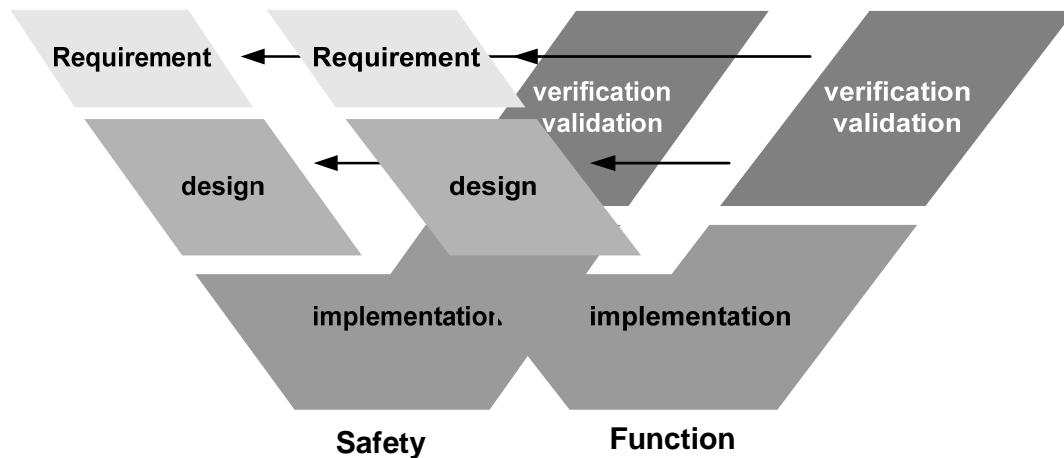


Figure 3-17: “Double-V-Model” according to [Ben04]

3.5.2.3 Safety norms in automotive industry

As have mentioned in Subsection 1.1, one of the most important factors for innovation of automotive electronics is the increasing requirements of customers for more safety and comfort, another import triggering factor is the ever strict requirements of industry norm, regulations and law makers about product liability, safety and emission to environment, etc.

In the following, state of the art of safety norms used in automotive industry will be briefly explained, since they play a significant roll in the appropriate steps in the development process.

3.5.2.3.1 *Automotive SPICE*

The Automotive Special Interest Group of the Procurement Forum is developing guidance on applying SPICE processes to the assessment of automotive software suppliers based on CMMI/ISO 15504 [ISO15]. A number of European automotive OEMs are already requiring that their software suppliers be subject to SPICE audits, and the intention of Automotive SPICE is to produce a common assessment framework that is acceptable to all the OEMs. Thus, if a supplier has been assessed according to Automotive SPICE and found to have a certain capability level, this is then acceptable to any OEMs, who require a supplier with a capability up to that level.

3.5.2.3.2 *Automobile MISRA Guidelines*

MISRA Guidelines (ISO TR 15497), better known as “Development Guidelines for Vehicle Based Software” was developed in the early 1990s by a UK-based consortium of automotive companies representing vehicle manufacturers, the supply chain and researchers. The guidelines were written against the background of the development that was taking place on IEC 61508 and acknowledged that, while the track record of the automotive industry in respect of embedded software was good, a recognized industry position on embedded software development for safety-related vehicle control systems was needed.

The notable approaches in the MISRA Guidelines are:

- The emphasis on safety activities lies in the development lifecycle.
- The use of “controllability” to classify hazards. This is a different approach from IEC 61508, although a similar process is used by the aviation industry.
- The use of (safety) integrity levels to classify systems and/or functions according to the level of risk mitigation required.

Although the MISRA Guidelines have been in use worldwide in the 10 years since their publication, a number of issues can be identified:

- There is no formal mapping for IEC 61508, despite many of the underlying principles being present.
- They do not explicitly address recent technology developments such as model-based development and automatic code generation.

3.5.2.3.3 *IEC 61508*

The international standard IEC 61508 relating to the functional safety of electrical, electronic and programmable electronic systems (E/E/PE) is now, also in the automotive sector, the generally recognized and most state-of-the-art basis for the development of safety-related systems. [IEC01]

Since IEC 61508 origins from the industry sector of process automation, it was developed against the model of “equipment under control”, for example an industrial process in a chemical plant; and of the safety functions that need to be applied to that equipment. These safety functions may be part of the control system for that equipment, or they may form a dedicated protection system.

IEC 61508 sets requirements for the reliability of the safety functions, which is known as safety integrity. In practice, safety integrity is classified into discrete Safety Integrity Levels. Hazard analysis is used to identify the safety functions that are needed, and the SIL of the system that performs them can be determined by classifying the severity of the hazards. The SIL indicates the reliability that must be achieved – where this can be calculated, for example, for electronic hardware; or the degree of rigor that must be achieved in the design and implementation measures in order to gain adequate confidence in the system. The definition of SIL in IEC61508 is shown in Table 3-1, where the SILs are specified with two categories of threshold, one for the probability of failure to perform its designed function on demand (dimensionless, for low demand system such as airbag), the other for Probability of one dangerous Failure per Hour (PFH) (for system with high demand rate such as steering/braking system)

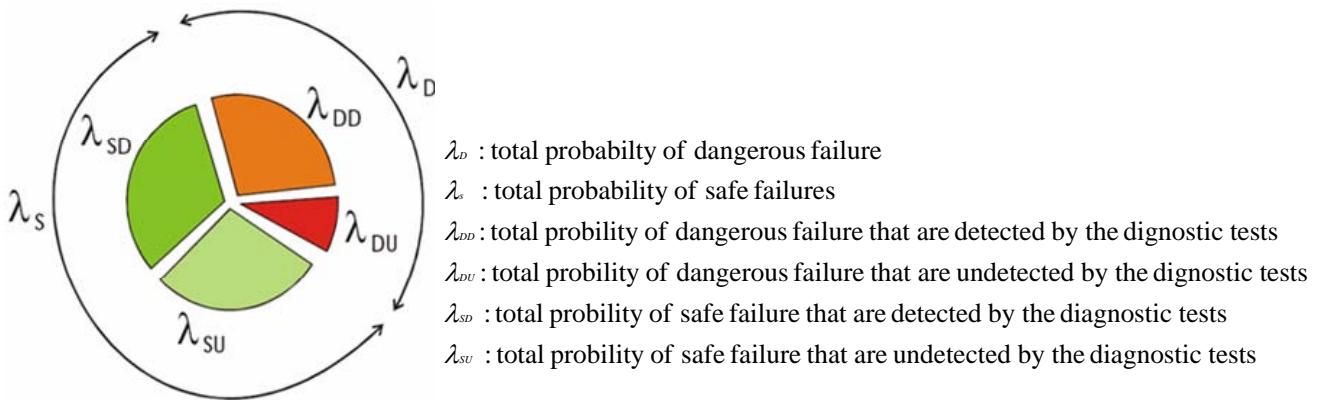
Safety Integrity Level (SIL)	Low demand mode of operation (Average probability of failure to perform its designed function on demand)	High demand or continuous mode of operation, PFH (Probability of one dangerous failure per hour)
1	$\geq 10^{-2} - <10^{-1}$	$\geq 10^{-6} - <10^{-5}$
2	$\geq 10^{-3} - <10^{-2}$	$\geq 10^{-7} - <10^{-6}$
3	$\geq 10^{-4} - <10^{-3}$	$\geq 10^{-8} - <10^{-7}$
4	$\geq 10^{-5} - <10^{-4}$	$\geq 10^{-9} - <10^{-8}$

Table 3-1: SIL definition in IEC 61508 [IEC01] (Part1 Subsection 7.629, pp. 76)

Another important feature for the classification of SIL is the definition of Diagnostic Coverage (DC) and Safe Failure Fraction (SFF) as shown in Figure 3-18.

$$\text{Diagnostic Coverage; } DC = \frac{\sum \lambda_{DD}}{\sum \lambda_D}$$

$$\text{Safe Failure Fraction: } SFF = \frac{\sum \lambda_S + \sum \lambda_{DD}}{\sum \lambda_S + \sum \lambda_D} = \frac{\sum \lambda_S + \sum \lambda_D * DC}{\sum \lambda_S + \sum \lambda_D} \frac{\sum \lambda_S + \sum \lambda_{DD}}{\sum \lambda_S + \sum \lambda_{DU} + \sum \lambda_{DD}}$$

**Figure 3-18: Calculation of Safe Failure Fraction according to IEC61508**

As shown in Table 3-2, depending on the number of tolerable faults, different classification of SIL required a certain threshold of SFF to be reached. Thus SFF is a key parameter for the safety case.

Safe Failure Fraction	Number of faults to be tolerated		
	0	1	2
< 60%	Not considered	SIL1	SIL2
60% - 90%	SIL1	SIL2	SIL3
90% - 99%	SIL2	SIL3	SIL4
=> 99%	SIL3	SIL4	SIL4

Table 3-2: Mapping of SIL to the SFF according to IEC61508

As defined in Figure 3-18, it can be easily proved that $SFF \geq DC$, thus DC is an important factor to improve SFF. That is to say, if we can provide the DC of a system is for example, higher than 90%, the SFF of the system is then higher than 90%, which is required for a SIL2-application.

Safety lifecycle as an important term was also introduced in IEC61508, which means the necessary activities involved in the implementation of safety-related systems, occurring during a period of time starts at the concept phase of a project and finishes when all the systems are no longer available for use [IEC04].

IEC 61508 is already being used as the standard to apply to automotive safety systems. The IEC 61508 “frequently asked questions” [IECFAQ] suggests “automobile indicator lights, anti-lock braking and engine-management systems” as examples of safety-related systems. However there are a number of issues with the direct application of the standard in the automotive domain:

1. The term “functional safety” in IEC 61508 only refers to the safety of the equipment under control and its control system. In many automotive systems, the manufacturer needs to be concerned with the safety of the system due to its control functions, not only the “safety” functions.

2. Many of the techniques and measures recommended in IEC 61508 are very specific to the industrial process control sector.
3. The model of validation in the standard does not align with the automotive industry practices of prototype development and testing.
4. Design and testing of embedded systems are not treated sufficiently.
5. OEM and supplier relations are not described in the IEC 61508 at all.

There are many useful techniques and concepts in IEC 61508, and the importance of this standard can not be underestimated. Nevertheless, further significant work is needed to adapt it to the automotive domain.

3.5.2.3.4 ISO WD 26262 from FAKRA working group

For the constraints of IEC 61508 mentioned in Subsection 3.5.2.3.3, there are now two European initiatives to implement automotive version and interpretations of IEC 61508. In Germany it is the DIN FAKRA committee AA13 and in France the BNA 0315B committee. The collaboration between the work forces of FAKRA, BNA and MISRA enables an integration and harmonization of the results into a new ISO standard ISO/WD 26262 (Working Draft, official release not expected before 2009) for the automotive functional safety.

This international standard [ISO24]:

- adopts a customer risk-based approach for the determination of the risks;
- provides a specific automotive analysis method to identify the safety integrity level of each undesirable effect (means for a vehicle function, identification of the consequences of one or some failures, leading or possibly leading to a customer claim, or a damage to the environment, up to significant damage or harm);
- uses automotive safety integrity levels (ASIL) for specifying the target level of safety integrity for the safety concept to be implemented by the E/E safety related systems;
- provides requirements for the whole lifecycle of E/E (engineering, production, operation, maintenance, decommissioning) necessary to achieve the required functional safety for E/E which are linked to the automotive safety integrity levels.

In this subsection the ongoing activities in FAKRA working group and main issues in the initial draft of ISO WD26262 will be briefly explained:

- The draft structure of ISO 26262 covers, as shown in Figure 3-19, the whole automotive life cycle, which consists of the following parts, under the general title “Road Vehicles - Functional Safety”:
 - Part 1: Terms and abbreviation [ISO21]
 - Part 2: Management of functional safety [ISO22]
 - Part 3: Concept phase including system definition, hazard analysis and risk evaluation and safety concept [ISO23]
 - Part 4: Product development System [ISO24]
 - Part 5: Product development: Hardware [ISO25]
 - Part 6: Product development: Software [ISO26]
 - Part 7: Production and operation [ISO27]
 - Part 8: Supporting processes [ISO28]

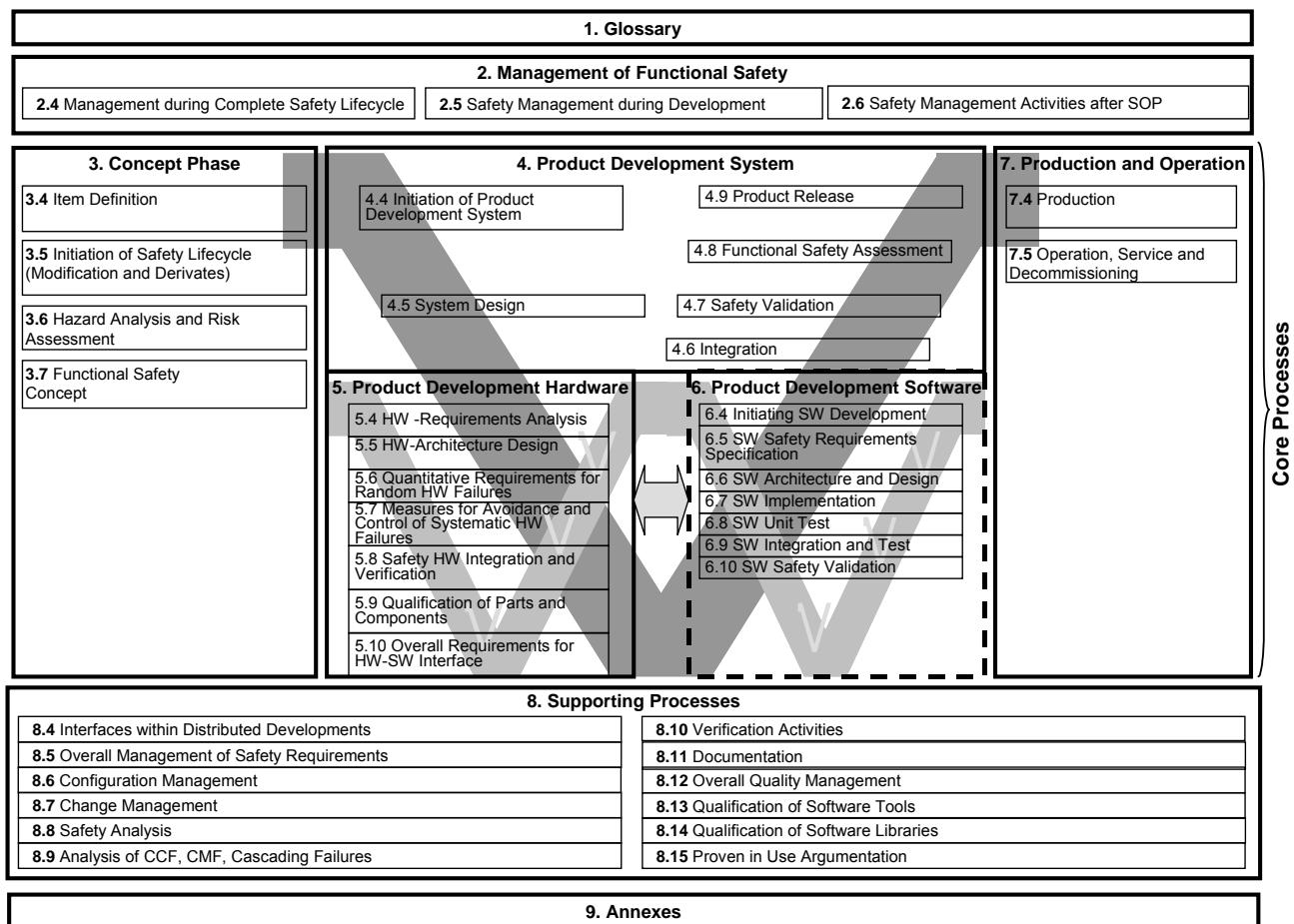


Figure 3-19: Overall framework of the safety lifecycle in ISO26262 [IEC01]

The overall framework of ISO 26262 considering the product safety lifecycle with the detailed activities in each phase is depicted in Figure 3-19, in which the shaded V-areas indicate important relations between various parts.

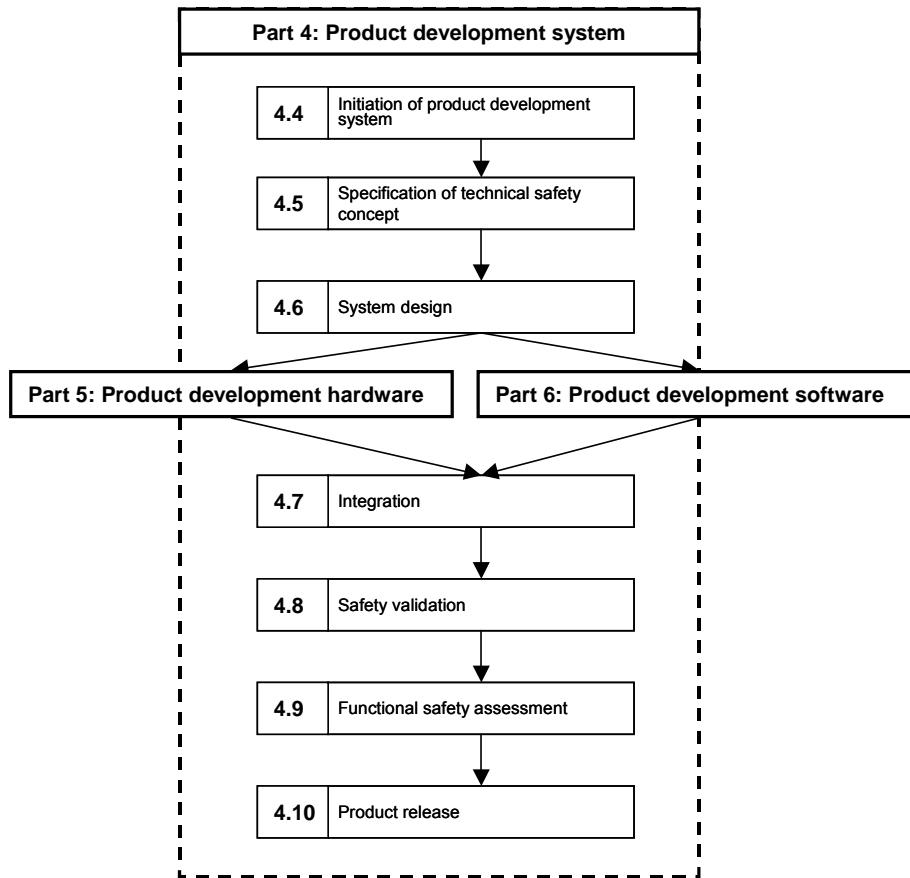


Figure 3-20: Reference phase model for the development of a safety-related item

The reference phase model of ISO26262, which is integrated in ISS Engineering Process in Chapter 7, is shown in Figure 3-20.

- Definition of ASIL: An important aspect in the activities of FAKRA is the derivation of functional safety integrity level, where the controllability of the hazard (controllability by the driver) is considered, where R = Risk is defined as

$$R = S \times f \quad \text{where } S = \text{Severity of damage}; f = \text{Frequency of occurrence}$$

And the frequency of occurrence f will be calculated by the formula:

$$f = E \times C \times r \quad \text{where } E = \text{Exposition} [-], C = \text{Controllability} [-] \text{ and } r = \text{Failure Rate}$$

According to the risk level, the safety integrity level scale devised in ISO26262 has four levels {ASIL A, ASIL B, ASIL C, ASIL D}.

Figure 3-21 shows a preliminary mapping between the IEC 61508 and ISO/WD 26262 scales. The mapping of ASIL-C can be chosen to the SIL-2 and SIL-3, while ASIL-D is only mapped to SIL-3 with PFH of $10^{-8}/h$ to $10^{-7}/h$, that is to say, SIL-3 is further classified into ASIL-C and ASIL-D in automotive industry, while ASIL-D application should have a $\text{PFH} < 10^{-8}/h$, so that it can fulfill the ISO-requirement anyway. In the following of this dissertation, ASIL will be used instead of SIL.

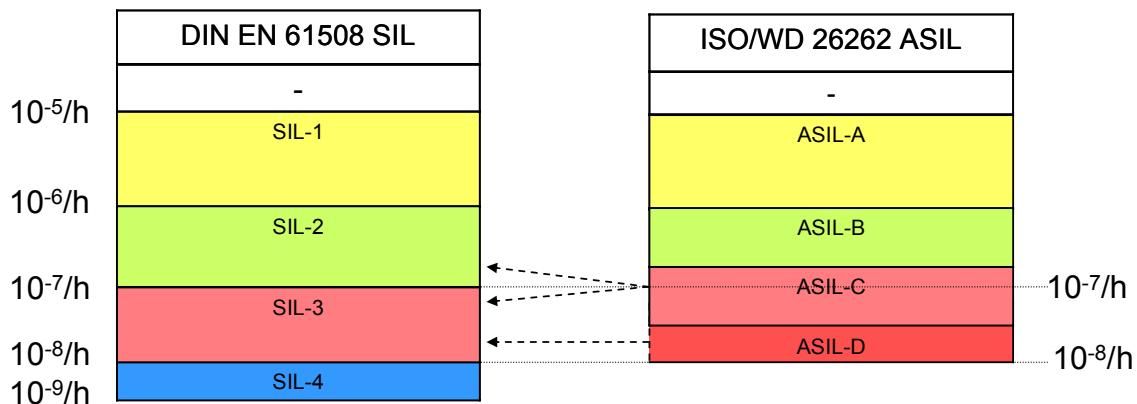


Figure 3-21: Preliminary mapping of safety scales between IEC 61508 and ISO 26262

- Safety Life Cycle: The safety life cycle in IEC61508 is not designed for vehicle as “consumer goods” but originated from process and automation industry, e.g. validation after installation etc. In consequence typical automotive test processes before SOP are not considered as for example HiL-tests, fleet tests and user oriented on-road tests. The activities of past SOP such as operation, maintenance and decommissioning are missing as well. The draft structure of ISO26262 is grouped into three large phases:
 - Concept phase
 - Vehicle development and production
 - Post SOP phase
- Thus the whole safety life cycle of vehicles is considered.

3.5.3 Conclusion for development process

The current development processes for automotive electronics tend to correspond to an iteration and adaptation of V-model. The simulation and model based development process reflects the trend towards increasing requirements for an early design validation, in order to avoid additional cost and time delays in case of system modification in late phases.

The development of safety relevant systems requires a dependability-justified process. Concepts of Hedenetz and Benz reflect this trend, in which safety analysis during the development of safety electronics with fault-tolerant mechanisms and consideration of safety aspects through the product life cycle, are integrated into the V-model. According to the state of the art in the safety norms, especially the future safety norms ISO/WD26262, dedicated safety steps should be carried out according to the automotive safety levels.

The trend for automotive safety electronics is heading towards freely distributed applications with high dependability, using standardized, configurable, fault-tolerant software and hardware architecture. These new requirements of integrated safety electronics cannot be fully supported by the existing solutions of development process.

4 Assessment of state of the art approaches and conclusion for challenges

In the chapters above, a general overview of the state of the art for automotive electronic architectures has been given. Current concepts and solutions for hardware architecture, software architecture and development processes of automotive electronics, taking dependability requirements into consideration, have also been introduced. In this chapter the assessment of different approaches from state of the art against new requirement and challenges of future automotive safety electronics (the so-called gap-analysis or delta-analysis) will be concluded.

- *Challenges in higher dependability despite of increasing system complexity*

There are already many automotive safety relevant applications based on distributed ECUs. In these applications, sub-functions or signals, which have previously been considered non-safety relevant, could become safety relevant for ISS. For example, the seat adjustment in the Pre-Safe system, the trail light control from the cabin for the brake assistant function and telematics function for calling medical rescue in the after-crash system can become also safety relevant in ISS.

Compared with state of the art safety relevant systems in vehicles, the Integrated Safety System is characterized by an even higher complexity, in which the implementation focus shifts from traditional mechanics and hardware to SW intensive systems. The timing of the cooperation between the distributed applications, which is relatively easy on one ECU, is undertaken by means of a communication network in ISS beyond domain borders. Thus an automotive ECU topology with fault tolerant communication, which supports the requirements of ISS in a cost effective manner is needed.

In future ISS-applications, X-by-Wire (steering, braking and engine control applications) without mechanical backup will be included. The higher dependability requirements for this (the ISS systems should be at least as reliable as traditional mechanical system, thus from fail-silent/fail-safe to fail-operational) can be only implemented with fault-tolerance characteristics.

As a summary, current E/E-architectures are unable to handle the required system complexity with higher safety requirements introduced by the ISS.

- *Challenges in the hardware architecture of automotive safety electronics*

With regard to ECU hardware architecture, a more and more complex in-vehicle network topology with different communication sub-networks is evolving, operating with different velocities and protocols, interconnecting ECUs and different automotive domains via gateways. With the introduction of new applications, there is a clear trend towards synergy effects by means of data fusion and more and more data flowing over domain borders. Safety relevant applications over domain borders, however, require deterministic behavior. QoS in communication is needed in both data integrity and timing. FlexRay, as the first truly fault-tolerant time triggered communication bus, will most probably be chosen for safety relevant communication.

Owing to the rapidly increasing complexity of automotive electronic systems, the existing solutions in automotive electronic architectures have almost reached their limits, especially with regard to safety relevant functions and different degrees of fault tolerance (fail-silent, fail-safe, fail-degraded and fail-operational).

Concepts for fail-operational ECUs with triplex or dual-duplex processors have already been given. However, these concepts must be adapted to the specific requirements for automotive applications, for example the optimization of complexity. The concepts for fault-tolerant ECUs should also be standardized and unitized with regard to the different requirements of safety integrity levels, to reduce development time and costs.

- *Challenges in the software architecture of automotive safety electronics*

In the ECU software architecture, there is a clear trend towards standardization and implementation of standard underlying software architecture (middleware), for the transparency of hardware implementation and application development, which will certainly improve the testability and dependability of ECU-software.

In the development of ECU software for safety relevant applications, the same trend is also apparent. Although software concepts concerning fault-tolerant mechanisms in ECUs already exist, there is no real fault-tolerant middleware architecture with standard dependability software modules to support common fault tolerance services (HW-transparent fault tolerance). As well as the basic standard software modules, some standardized software modules to provide common safety services shall also be defined. The different safety integrity level of distributed functions, implemented on several ECUs, requires configurable and fault tolerant software architectures to guarantee sufficient dependability while managing the system complexity and development effort. Fault model, especially in ECU software and containment region in software architectures are further important issues.

- *Challenges in the development process of automotive electronics of Integrated Safety System*

As previously mentioned, in the state of the art, traditional development processes and current simulation-based development processes have reached their limits in terms of managing the complexity and providing a full guarantee of the dependability requirement.

The new requirements in ISS with regard to distributed development and system integration should also be embedded or reflected during the whole development process, in which the safety requirements can be traced and validated. A certain safety integrity level should be supported during the system integration and validation phases.

In a development process, which fulfill both future safety norms (ISO/WD26262) and manages the challenges of ISS, standardized dependability services in both ECU software and hardware architectures should be reflected during the life cycle of the Integrated Safety System.

5 Introduction to EASIS project and EASIS approaches

The EU project EASIS (*Electronic Architecture and System Engineering for Integrated Safety Systems*) (see www.easis.org) was initiated in the year 2004 and finished in the first quarter of 2007. In the form of an industry partnership, EASIS is composed of 7 European OEMs, 8 system suppliers, 4 tool producers and other research institutes. EASIS is pursuing the goal of providing enabling technologies for the introduction of integrated safety systems, by developing an open and standardized dependable in-vehicle fault tolerant electronic architecture framework (reference model in both dependable software and hardware platforms) and a standardized systems engineering approach for integrated safety systems with a dependable E/E-platform.

This modular E/E architecture consists of two main branches: i) Software platform; and ii) Hardware platform. The software branch defines a common platform for software-based functionality, upon which future applications can be built, and the hardware branch defines a cost-effective on-board electronic hardware infrastructure. In this project, requirements and needs concerning ISS have been collected which constitute the basis for the definition activities for the two branches mentioned above [TRA06].

The EASIS work group on the software architecture designed a standard software platform for the execution of ISS, satisfying both ISS and external requirements, such as standards and hardware architecture. It will provide a basis for future in-vehicle electronic systems and include principles for software topology, an interface between hardware and software, basic fault tolerance and diagnosis mechanisms and inter- and intra-domain gateway services. The main composition of layers and functional areas of the software platform will be explained in the Subsection 9.2 in details.

A hardware architecture, which supports the high ISS demands regarding dependability, computational power, high speed, accurate information exchange and the software layers, will also be produced. This architecture must be scalable for standardized usage in both safety and non-safety applications, able to adapt to different domains and vehicle classes, capable of handling various sensors and actuators, have well-defined, standardized interfaces, support fault tolerance, error detection and error handling and have optimized costs and reliability.

Achieving system dependability will be much challenging for integrated safety systems than for traditional safety-critical automotive subsystems due to higher content of safety-critical software, higher complexity, and higher interaction of subsystems from different suppliers. The state of the art of automotive dependability methodology has to improve significantly to cope with these challenges. EASIS will provide guidelines for all major dependability-related activities. These guidelines are structured with the help of a simple dependability activity framework as shown in Figure 5-1.

The introduction of ISS will increase the networking needs not only for signals. Different ISS functions may work in the car thus making functional coordination and arbitration necessary. Connecting the vehicle to surrounding information infrastructures may influence the hard real-time chassis or powertrain control systems even more. All in all, the well understood basic-function-control-loop is transformed into an ISS-control-loop, see Figure 5-2. ISS-control-loop design requires new development processes and supporting tool chains, which the work on processes and tools will provide. A software specific systems engineering process (EASIS Engineering Process, EEP) suitable for the automotive industry, will be defined, integrating the results of system dependability, and a tool chain recommendation produced.

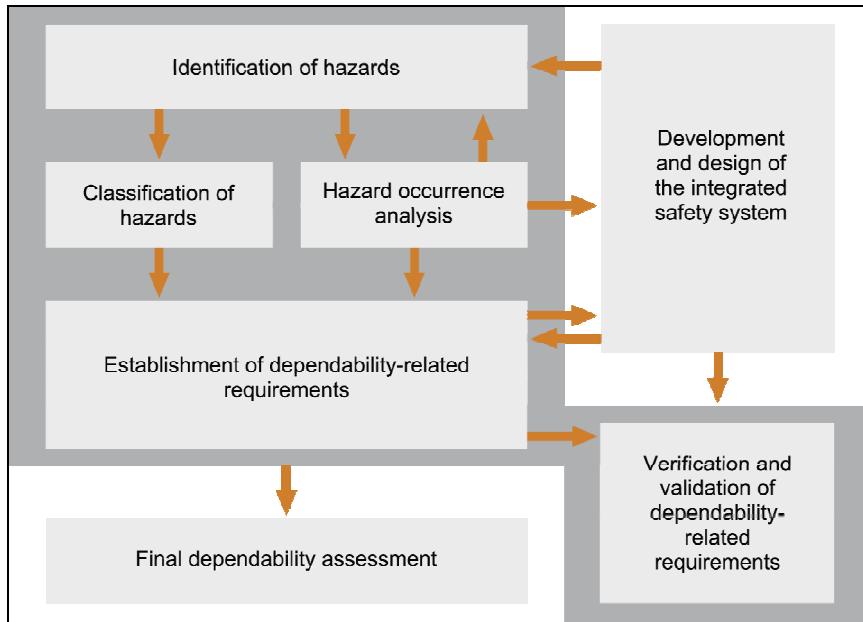


Figure 5-1: Framework of dependability activities

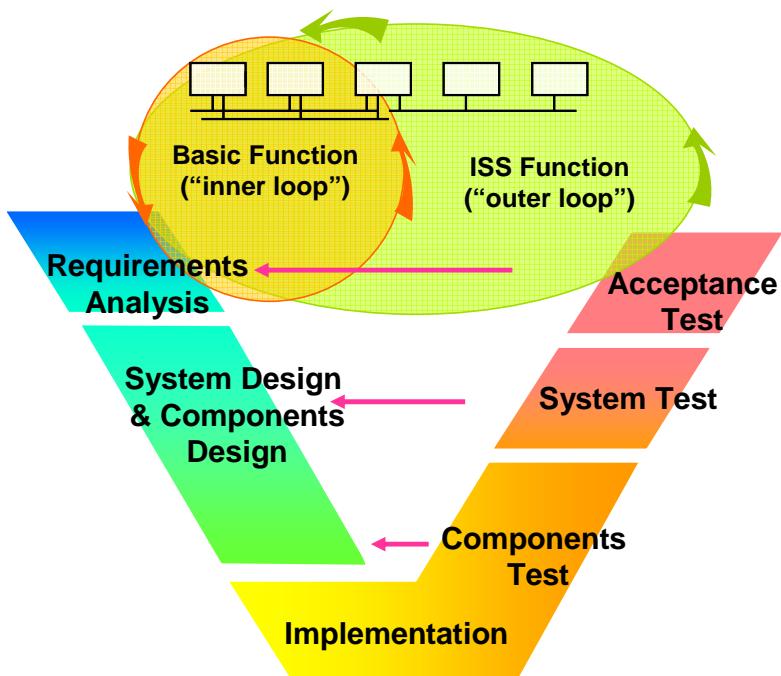


Figure 5-2: V-model indicating inner and outer loop

In the following sections the different work packages of EASIS-project will be briefly explained.

WP1: Software Platform

The primary drivers during development of the software platform are the main dependability attributes of safety, reliability, availability and security. Based on this, the work on dependability in the software platform is performed addressing the following three aspects: i) detection and

handling of faulty and erroneous states, i.e., fault tolerance aspects. ii) management of information concerning the state of the system, e.g., fault codes and internal fault information; and iii) consistency and integrity of data over the telematics link. Based on these three aspects dependability services are provided by the EASIS software platform.

WP2: Hardware Platform

The work on the hardware architecture focused on the following four aspects: i) logic system architecture composed of multiple ECUs; ii) communication networks connecting the ECUs; iii) internal ECU hardware architecture; and iv) power supply network. Based on these aspects, topics as Gateway, system topology, building blocks Network technologies, ECU hardware architecture and power supply were covered.

The hardware platform and software platform have been defined to closely integrate with each other, such that the services provided to the application components of ISS will have access to a wide selection of services.

WP3: System Dependability

The goal of the work package “system dependability” is to support the system engineering of integrated safety systems by providing guidelines for all major dependability-related activities in system development. The EASIS guidelines are structured with the help of a simple framework of dependability-related activities in system development and focus on the following three aspects: i) a comprehensive approach for the establishment of dependability requirements based on hazard identification, hazard classification and hazard occurrence analysis; ii) verification and validation of complex distributed control systems; and iii) means for demonstrating that safety was sufficiently considered in system development (safety case construction).

WP4: Processes and Tools

The work on processes and tools is focused on the following three aspects: i) systems engineering processes for ISS and their functional software components; ii) tool chains supporting the engineering processes; and iii) certification and test activities. Based on these aspects, an analysis of requirements (collected within EASIS and from other projects) was conducted that revealed new challenges for the development process that arise with the introduction of ISS: The shift from traditional hardware to software intensive systems, leads to a composition of safety functions within the vehicle that may be distributed across different ECUs. In case of diverging or conflicting signals or requests to actuators, the coordination / arbitration of different ISS functions will be a challenge, while requirements of system dependability will increase.

WP5: Validation

Key concepts and methodologies were validated in this work-package in order to show the applicability of these approaches to provide solutions to the problems and requirements identified concerning the development of integrated safety systems. For this purpose, two main validator set ups will be created:

- The main principles defined in the software platform and the hardware platform will be integrated into a validator resembling an automotive electronic system, including a telematics gateway, automotive sensors and actuators and several ECUs, all connected in a FlexRay network. In this validator, the feasibility and validity of the main principles of the software and hardware platforms are demonstrated.
- A safe speed function for use in commercial vehicles, automatically reducing truck speed to the maximum safe level for the road, by limiting engine torque and, if necessary, applying brake torque, will be created.

Concepts and designs in this dissertation were developed during my work as work-task leader in WP1, WP2, WP4 and WP5 of EASIS project, while the focus of the work lies in the work-tasks of software architecture, dependability software services, EASIS engineering process, tool chains and validation of concepts and methodologies.

6 Fault type and fault hypothesis for ISS

As written in Subsection 2.2.2.3, although fault model and faulty types analysis is not the focus this dissertation, the dependability requirements of a system are determined by its fault type definition and fault hypothesis. As mentioned in Subsection 1.1 and Chapter 4, future automotive Integrated Safety System, are facing quite a few new challenges. Thus we need to define a complete fault types and fault hypothesis for the ISS, which are mapped the development concepts of ISS in hardware, software design and development process.

The following subsections start with the introduction and motivation for fault hypothesis, the hardware and software.

6.1 Fault hypothesis – a major design step

A fault hypothesis states the assumption about the types and number of errors that a fault tolerant system must tolerate. It can be considered as a set of errors as $E_{FH} = \{e \mid e \text{ is a single error to be tolerated}\}$.

And with f_{system} denoting the functionality of a system S depending on a fault e .

$$f_{system}(S, e) := \begin{cases} \text{error-free} \\ \text{erroneous} \end{cases}$$

A fault tolerant system S is than defined as:

$$\begin{aligned} \forall e \in E_{FH} \\ f_{system}(S, e) &= \text{error-free} \\ \Rightarrow \text{the system } S &\text{ is fault tolerant} \end{aligned}$$

If the number of faults occurring in the real world exceeds the number assumed in the fault hypothesis, the assumption coverage will be low. Otherwise if the number of faults occurring in the real world is smaller than the number assumed in the fault hypothesis, the assumption coverage will be high. Even the best and most detailed fault model is useless if its fault hypothesis coverage is low.

A fault hypothesis can be used as a major design parameter for the system design and validation:

- Motivation for the design of fault tolerance algorithms: Without a precise fault hypothesis it is not known which fault classes must be addressed during the system design. Each introduced fault tolerance or self-checking mechanism has to be tailored according to the statements which are made in the fault hypothesis.
- Foundation for the System validation: As we know, a dependable system could not necessarily tolerate the faults, which it is not designed to tolerate, but it should be guaranteed that faults considered in the design phase should be able to be tolerated always. Thus the fault hypothesis is the key element for all kinds of validation techniques. For instances, the fault hypothesis can work as the description of the setup for fault injection experiments in an experimental validation. Different faults can result in the same or alike system errors. If the error can be tolerated by the system, it is most possible that the faults, which result in this error, can be also tolerated by the system.

It is to note that an exact fault hypothesis regarding number, source and temporal behavior with mathematic model are often not possible or realistic, esp. for the future software intensive embedded systems, taking the time constraints of the development into consideration.

In the following subsections, a general approach for the definition of a fault hypothesis suggested in will be used as a guideline [Kru98]:

- Partitioning o the system into SERs: A Single Error Region (SER) is a component of the system. It is assumed that faults will appear independent to each other and do not propagate to other SERs. The system can be partitioned into SERs by specifying these components by functionality plus the corresponding error.
- Failure semantic: Based on the previous step the failure semantic of SERs is defined. The failure semantic describes the behavior of a SER, in case of an error leading to a failure.
- Cause of failures: This step determines the maximum number of concurrently erroneous SERs and the SER internal causes. Thus, this step determines the actual requirements for the fault tolerance and self-checking mechanism of a SER.

6.2 Fault hypothesis of hardware in ISS

Starting from the fault hypothesis on a single ECU hardware structure, the fault hypothesis of communication networks, gateways and the other main hardware parts of integrated safety system will be defined in this subsection [DFP06].

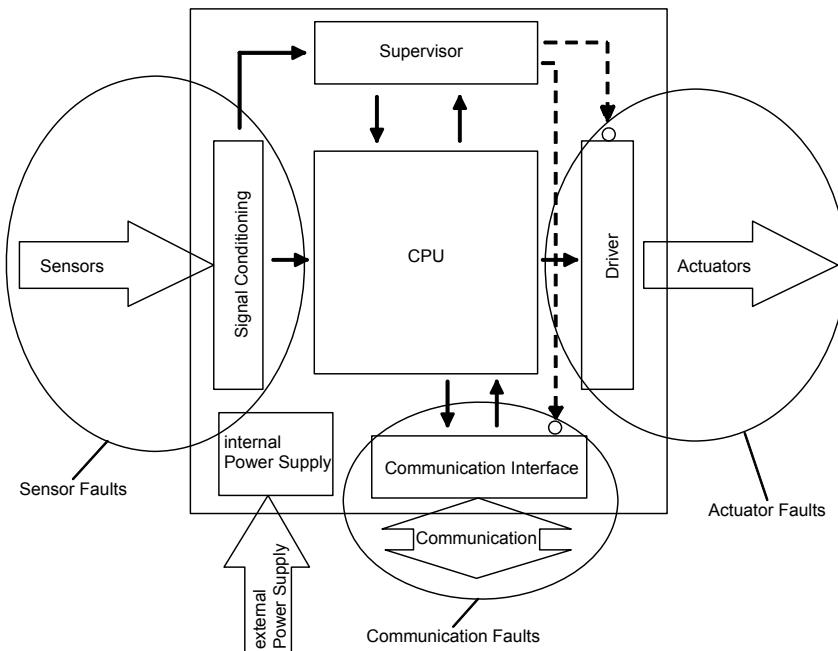


Figure 6-1: Simplified fault hypothesis of a standard ECU

As depicted in Figure 6-1, a very simplified fault hypothesis of an ECU can be divided into five categories:

1. Faults in the microprocessor/supervisor on the ECU
2. Faults by sensor coupling

3. Faults by actuator coupling
4. Faults by the power supply
5. Faults by communications

The 1st category of faults is listed as following:

Microprocessor core and supervisor:

1. Calculation errors (e.g. hardware error, logic error)
2. Value errors (e.g. hardware error, memory/register corruption, EMI, SEU, etc.)
3. Program flow errors (e.g. hardware error)
4. Interrupt errors (sequence, frequency, delay, disregarding, etc.)
5. Algorithmic errors (= Compiler/Logic Synthesizer errors / design faults)
6. Timing errors (error group V is not fully orthogonal to the other error groups)
7. Synchronization lost between CPU and supervisor
8. Supervisor and CPU are getting different information from the extern environment
9. CPU and supervisor use different, but both valid, rules to judge the control.

RAM/ROM:

10. Errors in the RAM/ROM (memory cell defective)
11. Faulty RAM/ROM access (wrong memory address)
12. Faulty memory mapping (=Compiler or linker errors / design faults)
13. Memory overflow

I/O-Interface:

14. Interface errors (errors in ADC/digital IO/ ...)

The 2nd category of faults concerning sensor coupling:

1. The sensor delivers no value or an error signal.
2. The read value of the sensor is wrong.
3. The sensor delivers a value with a wrong timing.

The 3rd category of faults concerns actuator coupling:

1. The actuator is not driven.
2. The actuator is permanently driven (without controller command).
3. The actuator is not driven at the right time.
4. The actuator is not driven with the correct performance.

The 4th category of faults concerning power supply can be divided into:

1. Over voltage
2. Under voltage
3. Short circuit
4. Over current (due to erroneously activated actuators, defective components, etc.)
5. Leakage current too high
6. Brown out (slow decrease of the supply voltage below the minimum limit)
7. Start-up timing
8. Shutdown timing

The 5th category of faults in the communication system between different nodes can be described as message/signal faults between the nodes. Faults in data exchange between the software-components within one node (intra-ECU communication) will be considered in the context of software faults.

Faults in lower levels, e.g. physical faults on a communication bus or communication interface can be mapped to the fault types listed below. At node level the following main faults will be relevant:

1. Data values of a received message are faulty (faulty data value)
2. The message is received later than a deadline (late message or message omission)
3. The message is received too early
4. The message can not be sent out in the given time window
5. The message can not be sent out

In distributed systems it is important to consider the fault propagation by distribution of message, which may stretch over several nodes and domain border. The faults can be subdivided into symmetric and asymmetric faults. The following main fault will be relevant:

6. All receiver of the message (in a special case only one receiver exists) regard the message as faulty with respect to the same main fault type, which is one of the fault types in 1 to 3.
7. All receivers of the message regard the message as faulty with respect to one of the main fault types 1 to 3, which can be different for each receiver.
8. Some of the receivers get a correct message, while the others get a faulty message with respect to one of the main fault types 1 to 3, which is the same for each receiver of the faulty message.
9. Some of the receivers get a correct message, while the others get a faulty message with respect to one of the main fault types 1 to 3, which can be different for each receiver of the faulty message.

Remark: The asymmetric fault types (7 to 9) are special cases of the well known “Byzantine General Problem” (as long as they are undetected), which is explained in details as following:

The replication of components, called structural redundancy, is very often a way to ensure that these systems are free from single points of failure. However, the use of redundancies within an electronic system introduces some undesirable effects.

Byzantine faults, also called as Generals Problem, was initially addressed by Lamport [LSP80] and [LSP82] more than 24 years ago. It refers to totally arbitrary failures, where the node/message may give contradictory information to different observers of the system, where Byzantine failure refers to the loss of system service due to Byzantine faults. i.e., arbitrary behavior in both the value and the time domain with inconsistent redundant values could be forwarded to different nodes. Generally speaking, there are two kinds of Byzantine faults:

- *Benign Byzantine fault:* A faulty node sends a message to a set of nodes. One or more of them do not receive a message at all.
- *Malicious Byzantine fault:* A faulty node sends a message to a set of nodes. One or more of them do receive a message, but with a wrong value.

In [SIV04], the Byzantine faults in embedded systems are revisited from a practitioner's perspective, where a few typical Byzantine faults are demonstrated.

- Digital signal that is stuck at "1/2" because of the physical nature of logic circuitry (some internal faults in the transmitter) as shown in Figure 6-2.

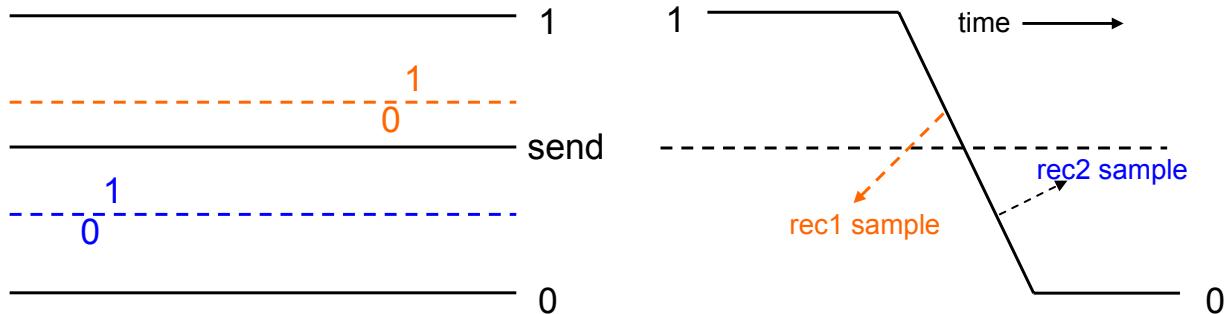


Figure 6-2: Cause of Byzantine fault with "1/2" area threshold

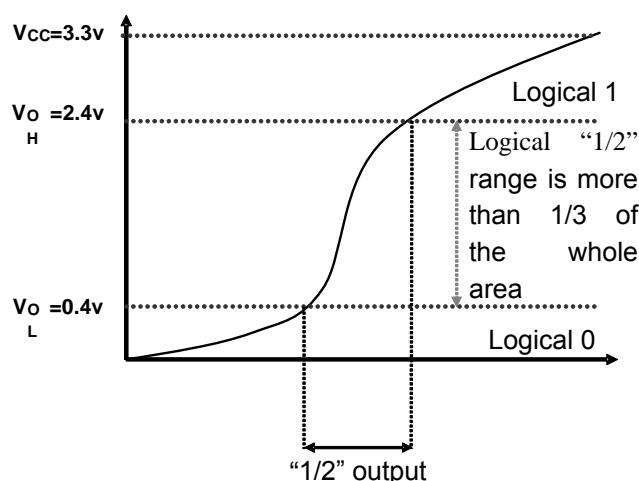


Figure 6-3: Gate transfer function with "1/2" area

Figure 6-3 shows a logic gate's transfer function for a common 3.3 volt logic circuit family. The transfer function shown here has a much less step slope than a typical logic circuits in order to show the 1/2-area.

- Byzantine faults may propagate through multiple stages of logic and still remain at an ambiguous level. Thus even if the transferred data is protected with CRC (e.g. Schrödinger's CRC), each CRC bit affected by the $\frac{1}{2}$ data bit can also be $\frac{1}{2}$. As shown in Figure 6-4, the resulting data received by a and b are different, but each copy has a correct CRC for its data as shown in Figure 6-4. Thus CRC can not provide a guaranteed protection against Byzantine fault propagation.

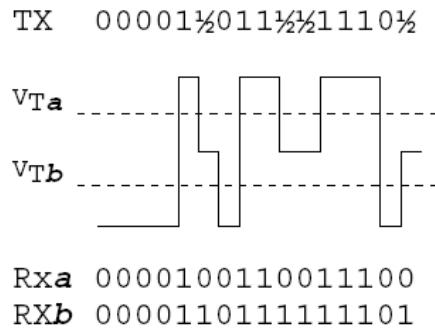


Figure 6-4: Byzantine Faults in Schrödinger's CRC [SIV04]

In time triggered communication systems with a deterministic fault tolerant communication protocol, Byzantine faults can not be excluded either. Firstly, because of EMC-problems and random radioactive decay, bit-flips in register and RAM can be induced. The dominant Byzantine faults are due to marginal transmission timing, that a message comes to the receiver at the margin of the time cycle. In this case, for example, corruptions in the time-base could lead to the situation, that a message transmitted slightly too early are accepted by the nodes with slightly fast clocks, while nodes with slightly slow clocks will reject this message.

6.3 Fault hypothesis of software in ISS

While the focus of traditional fault hypothesis of in-vehicle electronics lies in the hardware architecture of ECU, an important part of fault hypothesis of Integrated Safety Systems is software relevant. Two main challenges by the development of ISS are the implementation of software intensive system and integration of functions with different safety requirements on one ECU.

- The introduction of ISS requires by its natural consensus from different ECUs. This required consensus can be low-level (e.g. time synchronization among the distributed nodes with inter-communication) or at a higher level (e.g. a coordinated interaction of distributed applications like distributed controlling of brake force in ESP or distributed fault tolerance on several nodes in a coordinated manner).
- Taking the supply-chain of automotive industry, while the responsibility of tier1 suppliers is shifting from components to the system solutions, the focus of OEMs are shifting to the system integration. The integration of software components from different suppliers or even products third-party pure software providers on the same ECU could inevitably introduce more efforts of integration by OEMs. Although improved software requirement specification, standardized software development process and software architecture with defined interface can help to ease the problem, a specific fault hypothesis of software will surely help in the validation and debugging phase of system integration.

As introduced in the state of the art in the Subsection 9.2 software architecture for the in-vehicle electronics, an appropriate fault hypothesis of ECU software platform with layered standard software services will be introduced as following.

6.3.1 Software timing faults

As state of the art of automotive electronics, an OSEK-conform operating system with pre-emptive scheduling is used, because it is assumed that not all tasks can be scheduled as periodic and that event triggered tasks are still needed. In an operating system with pre-emptive scheduling, according to a predefined scheduling algorithm (e.g. priorities) the scheduler shall execute the parallel running schedulable entities, in which scheduling faults can take place:

1. Activation during running process of other entity: During execution of a schedulable entity, it might be possible that other schedulable entities shall be activated. Due to the reason, that parallel processing is not possible, one of the schedulable entities can not run.
2. Deadline/execution time budget violation: Deadline/execution time violation occurs, when a scheduled entity can not fulfill its job within the predefined time budget.
3. A specific schedulable entity (e.g. non periodic task) is activated too often: During runtime a schedulable entity might be activated several times in the same time window. This in general leads to inconsistencies and errors, if that routine is not prepared to do so.
4. Maximal number of entity activation is reached: During runtime more schedulable entities might be activated than the scheduler implemented on the ECU can manage, e.g. too many tasks are scheduled and activated.

In general all these scheduling concerning the operating system faults can result in two final faults: missed activation and missed termination of the OS-objects.

The scheduling faults are mainly considered to be faults, which shall be avoided during the design and development phase of the system or functional unit, but this kind of faults can not be excluded since there is no 100% fault free software. Thus the scheduling faults shall be handled (detected and tolerated) by the dependability software platform in the runtime. Detailed fault hypothesis considering timing faults in the runtime will be discussed in Subsection 9.3.2.3.2 in the dependability software service of software watchdog.

6.3.2 Communication between SW-components

As mentioned in Chapter 3, a clear trend in the embedded systems, especially in the automotive industry is the distributed control system. Distributed control system, by its nature, require consensus and coordinated behavior among the constituent parts. The required consensus might be low-level, (e.g. time synchronization among the distributed nodes) or at a higher level (e.g. in form of coordinated distribution of brake force in ABS or ESP system).

Thus the communication link is a critical chain with strict dependable requirements. At level of software-components the following main faults or errors concerning data exchange and communication between software components on one ECU (intra-ECU communication) or on different ECUs (inter-ECU communication) will be relevant: Except for the five fault types listed in Subsection 6.2 about communication, API access fault is an additional fault to be considered here.

6.3.3 Concurrent resource access

As assumed, different application SW-Cs (or redundant SW-C) can be integrated onto one ECU, limited resource on ECU may be accessed concurrently by different entities. The resources here can be CPU running time, SW-services, peripheral or other resources.

For peripheral access, one physical unit/peripheral device (like an actuator) is requested to be controlled by two or more independent software-components. But in general it is assumed that a specific physical device is controlled by only one software-component at a certain time. This assumption does not exclude e.g. a possible configuration, that two different application SW-Cs control an actuator. For example, one unit can "enable" the actuator and the other one can control it. This means that the actuator works like an AND gate, which reduces the possibility of unwanted control. Faults can occur here during a simultaneous access (timing and duration) of a peripheral device in the control of peripheral drivers. The fault in concurrent resource access can mainly be considered as a fault which has to be avoided during system design phase. A device that can be accessed by several applications would have a guarding queue. Possible problems that can occur during a simultaneous access of a peripheral device should be checked and minimized during system design and integration.

7 Engineering Process of Integrated Safety System

As required by the safety norms in Subsection 3.5.2.3, development process according to the appropriate safety integrity level is the most important lever to face the challenges summarized in Chapter 4. In the following an Engineering Process for the development of future Integrated Safety System will be discussed.

7.1 Introduction to the ISS Engineering Process

In the Subsection 3.5.2, state of the art in the development process for automotive safety electronics and challenges introduced by development of new ISS-applications are introduced. The analysis of the gap shows the necessity to improve today development processes. The widely accepted safety norm IEC 61508 and its up-coming adaptation for the automotive industry, the industry norm ISO/WD 26262 should be taken into consideration. At the same time the system dependability should be improved by managing the complexity of ISS.

At the beginning of this chapter, an overview of the ISS Engineering Process as part of the global development process is given in Figure 7-1.

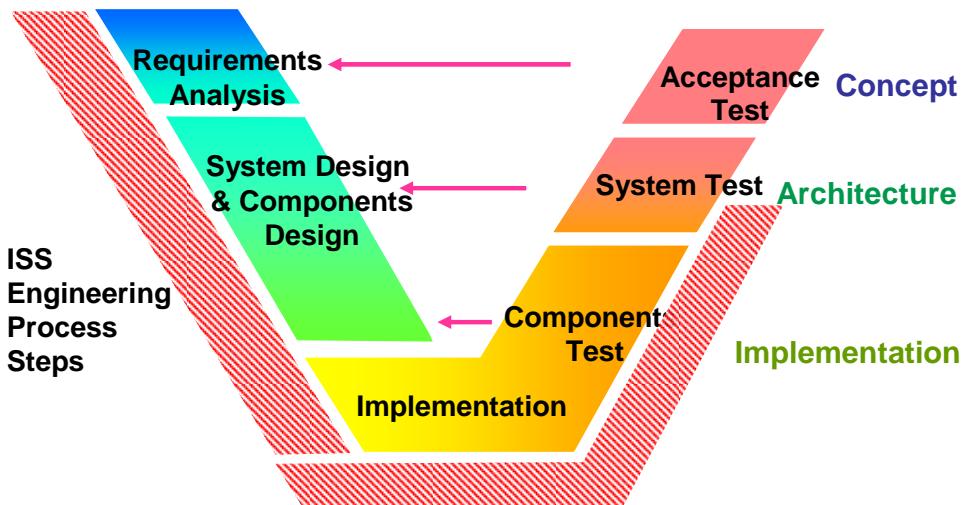


Figure 7-1: Overview of ISS Engineering Process with virtual front-loading

The complete development of ISS covers multiple phases from design to the serial production as shown in Subsection 3.5.2.2 as a series of V-models. The ISS Engineering Process focuses on the phases from A-sample to the B-sample, while keeping the consistent requirement engineering, safety analysis and system validation with the following phases of C-sample and D-sample.

The basic of the ISS Engineering Process is the V-Model'97. In the ISS Engineering Process, the intern V is extended with an extern V as the pink part shown in Figure 7-1. The extension of external V is based on the EASIS Engineering Process (EEP) proposed in the EASIS project [EAD41].

Compared with the EASIS EEP, the ISS Engineering Process integrates a consistent requirement engineering phase, which covers from the requirement analysis including hazard analysis to the component test and part of system test with fault injection. The requirements from ISO26262 are

joined together with the detailed development steps. The dependable software and hardware architectures, defined in the Chapter 8 and Chapter 9 as a trend of platform strategy, are combined in the ISS Engineering Process as well.

From the global view however, the emphases of the ISS Engineering Process are listed as following:

- Virtual front-loading: Strict analysis and validation activities are commonplace in today's automotive development processes. However, these activities are usually performed rather later – when major design decisions have already been performed and the effort of changes is relatively high. Moreover there is often simply insufficient time left to conduct complete system test. According to a McKinsey survey [McK05], about half of automotive electronics are tested in vehicle 18 months before SOP and only 6 months before SOP does the level reach 90%. The ISS requires an even more decentralized development process, a distributed rapid prototyping based on the principle of early integration and validation with advanced simulation environment is one of the enabling technologies to solve this problem. The physical integration of serial sensors and actuators with the microcontroller in vehicles can take place in the late phase of product development; however, a lot of safety features can be already covered and tested with virtual front-loading.
- Correct-by-construction approach: correct-by-construction means that development artifacts adhere to certain quality measures by following a strict development approach. The correct by construction approach can reduce development costs by automating development steps and reducing the analysis of the results. Furthermore, it can increase the resulting quality in terms of non functional requirements like availability and safety of systems. Thus it is a promising approach especially for the development of ISS.

Main benefits of using the ISS Engineering Process are:

- This process helps to handle the complexity by splitting up the system development into manageable parts without losing overview (as a strict development approach for correct by construction).
- This process supports the identification and further handling of safety related requirements with enhanced end-to-end requirement tracing. On every development step, the necessary safety activities are identified and triggered.
- The virtual frontloading of hazard analysis activities, model-in-the-loop, software-in-the-loop and hardware-in-the-loop test support the early detection of (design) errors. Problems with the overall concept (architecture) can be detected early. This offers the possibility to make the adjustments on concept level, and not on implementation level.
- The process takes the current trends of standardization of dependable hardware and software platform as well as safety norms into consideration.

Figure 7-2 shows in detail the domain of virtual front-loading in the ISS Engineering Process. It begins with the requirement engineering. After the FAA (Functional Analysis Architecture) is mapped to the hardware architecture, the FDA (Functional Design Architecture) model derived can be validated with the rapid-prototyping. The virtual front-loading accompanies these steps and checks the results against the stepwise improved and complemented requirements.

The correct by construction approach is implemented here with the exactly defined steps in the EASIS Engineering Process as well the mapping and interconnection between the steps, which improves the quality of the designed prototype. The virtual frontloading here, in return, can find out more potential deficiencies of the design and enables a prompt and cost-effective re-engineering of construction, which contributes to the correct by construction approach as well.

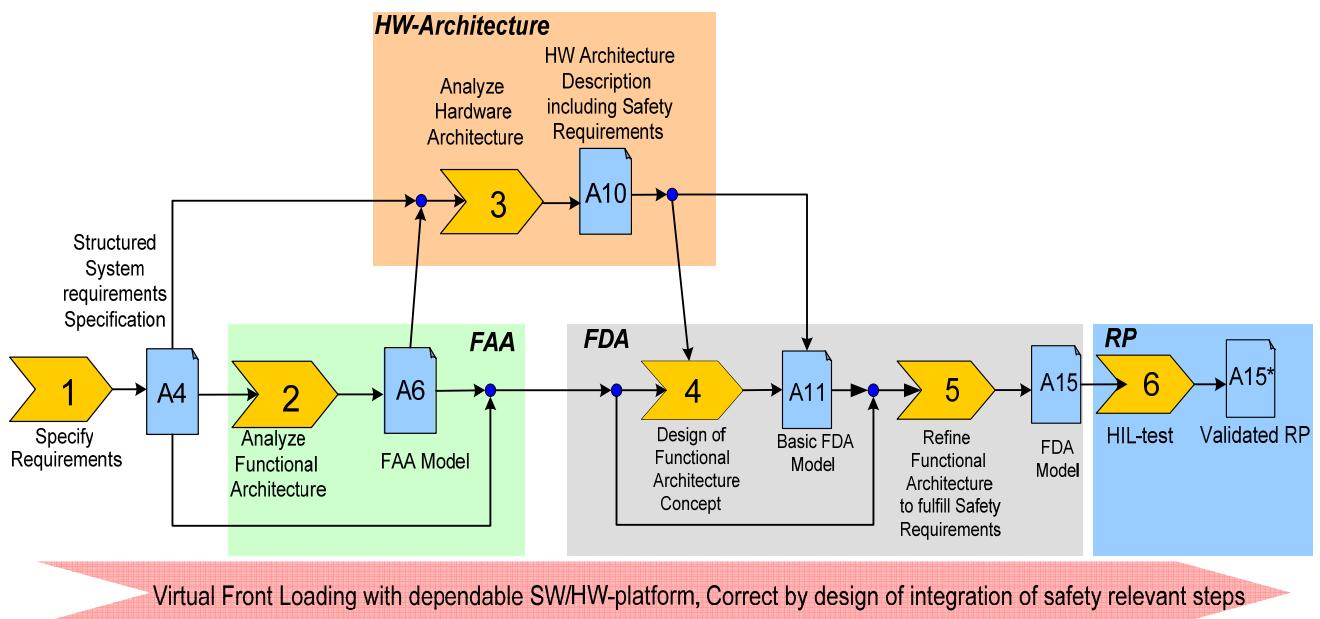


Figure 7-2: ISS Engineering Process with virtual front-loading in details

As depicted in Figure 7-2, the ISS Engineering Process can be divided into five main parts as following:

Part 1: Initial of requirement engineering (specify preliminary requirements)

All requirements of the system function under development are collected, specified and structured. Safety related requirements are analyzed from a system level point of view. At this step, it is not necessary to make assumptions regarding the E/E architecture, as mainly the intended functional behavior is considered and the possible malfunctions are analyzed.

Details on this process step are given in Subsection 7.2.1.

Part 2: Development of Functional Analysis Architecture

Based on the structured system requirements specification, the functional properties are modeled using the Functional Analysis Architecture. In this process part, firstly the functional architecture is

defined (defining the building blocks) and the functional interchange mechanisms (timing and dynamic behavior) are analyzed. Secondly, the functional behavior of the single building blocks is modeled.

In this stage, the basic cooperation of the function under development with the other components of the vehicle is analyzed. This minimizes the risk of late detection of basic incompatibilities between the specified behaviors of the different functions in the vehicle. The FAA defines the possible basic entities and interaction rules usable for the considerations on this abstraction level (notation and semantics).

The FAA model as elaborated in this step is analyzed e.g. consistency, timing and formal properties. The list of the identified FAA blocks can be used as basic input for dependability related analysis (e.g. FMEA). Virtual front-loading with model based simulation of functional behavior is used for validation as to the functional requirements. Following the dependability analysis, a first Hazard Analysis is performed, possibly generating new requirements and an update to the list of hazards. Details on this process step are given in Subsection 7.2.2

Part 3: Development of hardware architecture

With the FAA Model of the function system under consideration, it is possible to tentatively allocate the defined function blocks to one or more hardware entities (hardware architecture with layout of ECU units, sensors, actuators and communication network). Thus the Hardware Architecture (HA) is integrated here as an input factor. This enables the analysis of the dependability requirements that these hardware components have to fulfill, leading to a description of necessary redundancy and of the possible failure modes. This will lead to a refinement of the hardware architecture, and give a specification of failure reaction requirements. Details on this process step are given in Subsection 7.2.3.

Part 4: Development of basic Functional Design Architecture

The hardware architecture gives the basis for the next analysis and modeling steps. First, the final allocation or mapping of functional components (as defined in the FAA model) to the hardware architecture is performed, while also taking the safety constraints into consideration. The software platform for ISS including dependability software services and its impacts on the functional architecture model are added to the design of the functional architecture. In other words, the whole dependable software platform is integrated into the design.

Details on this process step are given in Subsection 7.2.4.

Part 5: Refinement of and validation of FDA model with SiL-test

The Basic FDA Model covers the behavior of the function under the assumption that no faults (hardware, software or communication) occur. In Part 5, the handling of faults according to the system fault model is integrated, giving a final FDA model including the safety concept of the function system under development.

Details of these process steps are given in Subsection 7.2.5.

Part 6: Hazard analysis and validation of FDA model with HiL-test

Hazard analysis will be carried out for the complete FDA model with the new information. By means of hardware-in-the-loop test with consistent evaluation cases on a hybrid-system of rapid-prototyping platform, evaluation microcontroller and rest-bus simulation method, an end-to-end tracing to the test specification and requirement specification can be guaranteed.

Details on this process step are given in Subsection 7.2.6.

The enabling steps for the development of ISS, along with safety relevant steps as extension of EASIS Engineering Process, are listed as following:

- Enhanced requirement engineering phase with seamless process oriented mapping and tracing of requirement specification (seeking “end-to-end” quality assurance).
- Using UML for the functional architecture design of the embedded system and the specification of interface and specification of dynamic behavior.
- Front-loading with the virtual integration, taking the current trends of hardware architecture and software platform standardization (results from Chapter 8 and Chapter 9) into consideration (early model-in-the-loop and software-in-the-loop test).
- Mapping of app. SW-Cs to electronic architecture taking safety requirements into consideration and a guideline for the mapping work.
- Application of dependable SW-platform and hardware platform in the system integration by configuration of dependability services.
- Verification and validation with rest-bus-simulation and hardware-in-the-loop test and fault injection based on fault hypothesis.

In the following chapters, the development processes with the appropriate tool (hardware and software) are given step by step in details after the landscape of the whole development process is shown. The notations and symbols used in the landscape of the ISS Engineering Process are shown as following Figure 7-3.

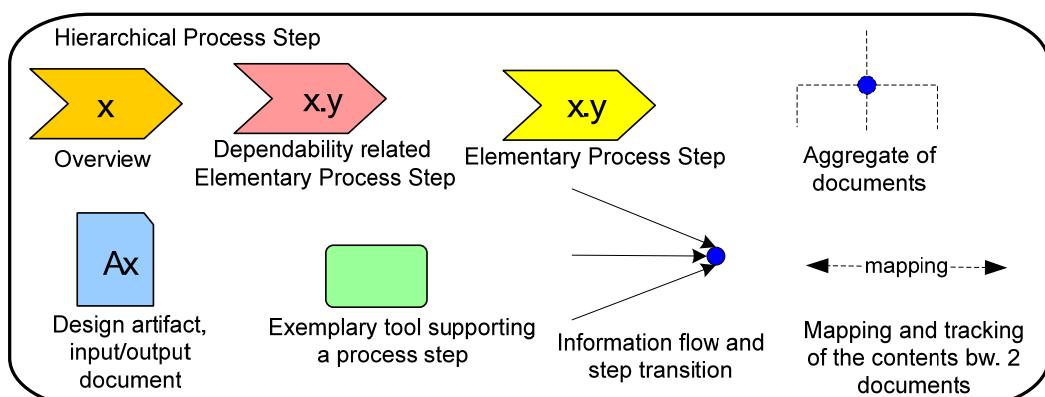


Figure 7-3: Notation in the ISS Engineering Process

7.2 Development steps of ISS Engineering Process

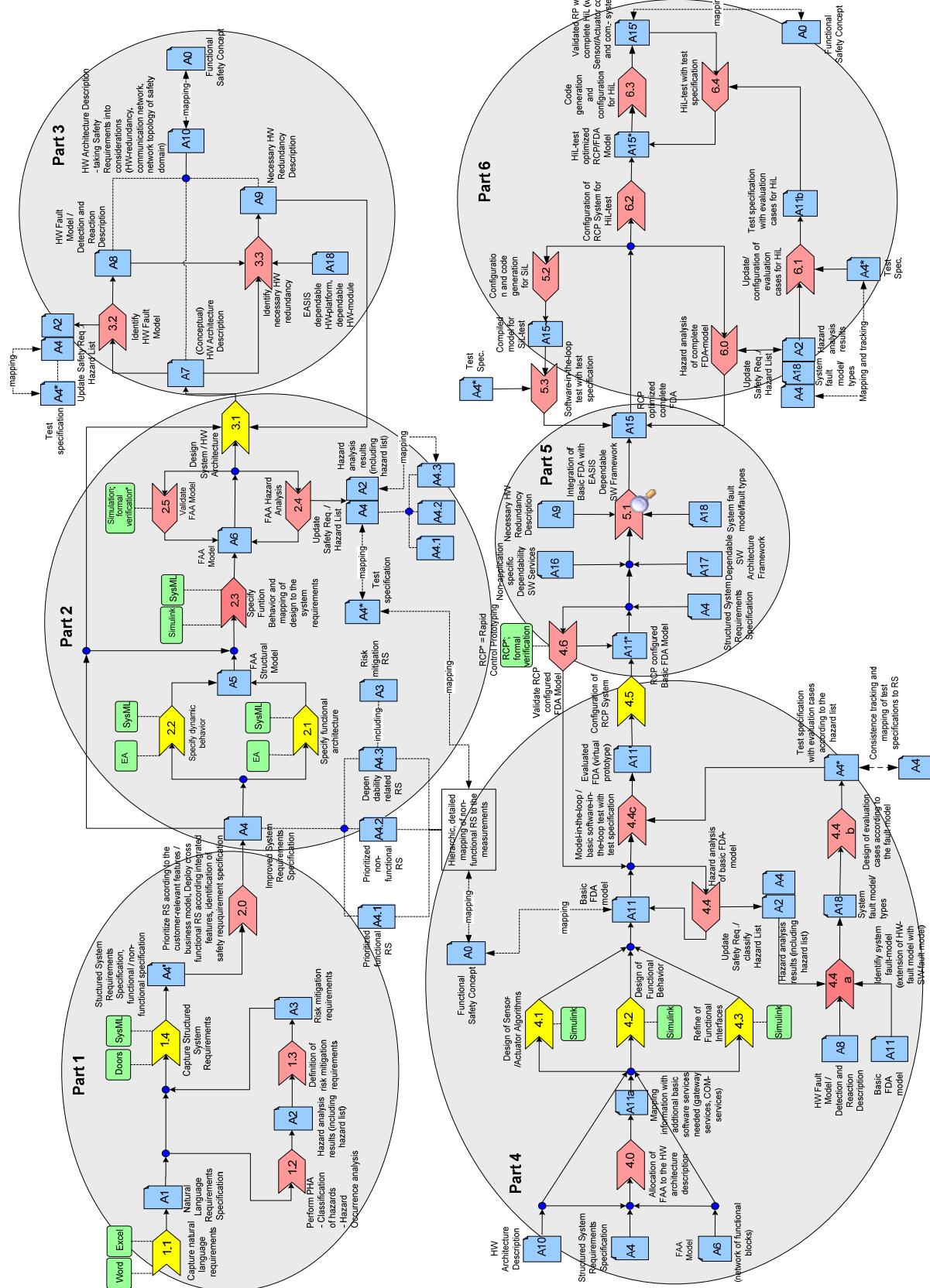


Figure 7-4: Global view of ISS Engineering Process

An overview of ISS Engineering Process is depicted in Figure 7-4, which is subdivided into 7 parts as the following subsections.

7.2.1 Part 1: Initial of requirement engineering (specify preliminary requirements)

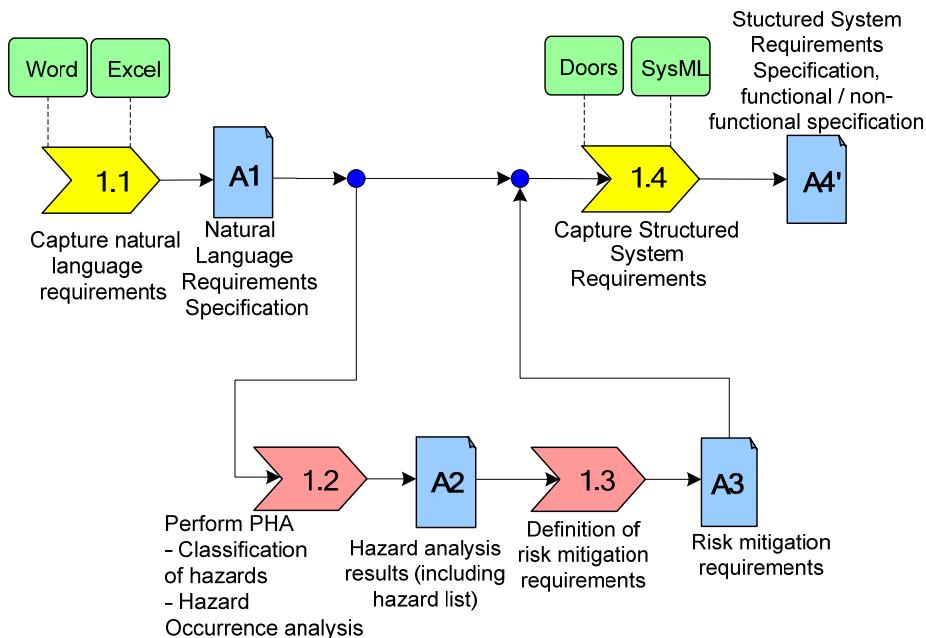


Figure 7-5: Specify preliminary requirements

Process step 1.1: Capture natural language requirements

The visions of a new system including system requirements and system concept are firstly collected in a requirement specification of natural language as depicted in Figure 7-5, where e.g. a high-volume vehicle manufacturer targeting the mass market cost-consciously strives to carryover concepts from previous projects. The next vehicle system concept is typically an adapted variant of previous one.

Already in this initial step two issues should be considered [McK06]:

- Applying right level of specification granularity – If the granularity is too high, there is the danger of misunderstanding. If too low, the efficiency can suffer.
- Deploy cross-functional feature specifications by setting up a cross-functional team to work on the specifications.

The proposed tools are more non-formal tools like Word, Excel and state-transition diagrams.

Process step 1.2: Perform Preliminary Hazard Analysis (PHA)

Preliminary Hazard Analysis (PHA) is the first execution stage of a product project specific plan, performed at the conceptualization stage of the ISS lifecycle. PHA does not require concrete designs. PHA identifies potential hazards leading to unacceptable risk, and specifies corresponding requirements to reduce the risk to within tolerable limit. As the ISS progresses

towards concretion and requirements flow down, the PHA may be iteratively applied to the corresponding subsystems. PHA is iterated until residual risks are within tolerable limits.

Measures to perform the process step:

- Identify losses to be avoided & corresponding severity category
- Identify hazards based on previous experiences
- Identify hazards based on concept review
- Categorize hazards by severity
- Assess risk of malfunctions (see remark ¹⁾)
- Tabulate hazard-loss combinations & estimate risk level
- Establish tolerable risk targets
- Perform Hazard Graph Analysis (HGA)

While the results of such a Preliminary Hazard Analysis are defined as following:

- (General) list of losses to be avoided
- Database of hazard-loss combinations, corresponding risk level
- (General) preliminary hazard list, including
 - Hazards from driver overload, based on workload and cognitive load model
 - Hazards from driver confusion about mode
 - Hazards with unacceptable risk levels

The proposed tools for the early stage of the PHA can be supported some form of templates, questionnaires and tables, e.g. in Excel. The hazard graph, as mentioned above, can be integrated with the state machine model.

Process step 1.3: Definition of risk mitigation requirements

This process step specifies requirements to reduce the risk of each hazard to within tolerable limit. As mentioned above, the PHA is iterated until residual risks are within tolerable limits.

Measures to perform the process step:

- Identify requirements for risk mitigation
- Perform Hazard Graph Analysis
- Identify preliminary verification and validation (V&V) requirements

Results of the process step:

- Preliminary requirements and constraints for risk mitigation
- Additional hazards introduced due to risk mitigation measures
- Analysis of newly introduced hazards

¹ According to ISO26262-3 [ISO23] Section 6 hazard analysis and risk assessment, Risk of malfunction = severity * probability. In this context, "probability" is defined as "the probability that the vehicle is in a driving situation, in which the malfunction could cause a mishap with the considered severity". Example: A possible malfunction of a gearbox is that the car does not start driving when intended by the driver. This is normally no problem, as a car that is standing is the safest. But if this malfunction happens when the car is standing on a railroad crossing (for any reason), this is not safe at all. For this situation, this analysis step analyses what percentage of vehicle's lifetime that it is in such a particular situation.

- Identification of newly introduced hazards
- Conflicts with other requirements, including trade-off analysis and conflict resolution
- Preliminary requirements for verification and validation
- Hazard graph after applying risk mitigation requirements. The risk mitigation here means both the reduction of the hazard probability and mitigation of the effects of hazards.

The proposed tools are again the templates, questionnaires and tables as used in step 1.2.

Process step 1.4: Specify structured system requirements

In this step the system safety requirements are integrated with the discovery of adverse relationship (conflict) with other requirements, thus some kind of trade-off preliminary decision should be carried out. The proposed tools for the requirements engineering including requirements managements and tracing are DOORS and SysML [SML06].

7.2.2 Part 2: Development of Functional Analysis Architecture (FAA Model)

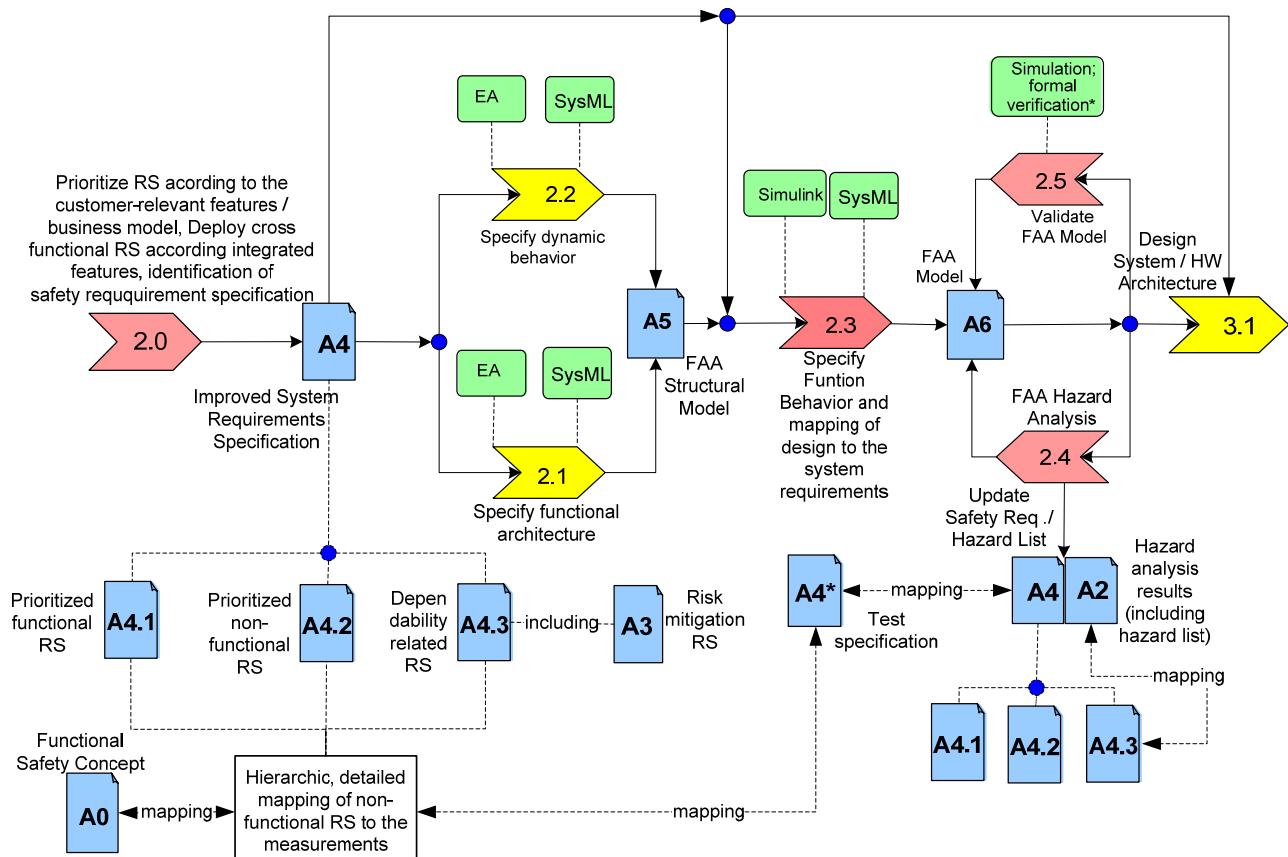


Figure 7-6: Safety integrated requirement specification and development of the FAA-model

Process step 2.0: Improvement of requirement specification

This process step (as depicted in Figure 7-6) improves the system requirement specification by further iteration of prioritization and categorization of the requirements according to different constraints.

Measures to perform the process step:

- Prioritization of the requirement specifications
 - The safety relevant requirement specifications, of course, should be given the highest priority (e.g. the allocation of ASIL according to ISO26262). The ISS, however, are mostly cross-functional integrated features. The system dependability requirement should be split to the sub-functional safety requirements.
 - Identify requirements for customer relevant features. In-vehicle electronics inject a lot of innovation in automotive industry, but not all the features can be perceived by the customers in proportion to the development efforts behind. With the introduction of vehicle platform strategy, based on one vehicle platform esp. on the vehicle E/E-platform, different vehicle variants according to the defined business model can be configured. Thus for each vehicle model, the individual customer relevant requirement specification should be selected and given a higher priority.
- Categorization of the requirement specifications
 - The easiest way is to divide the requirements into functional and non-functional requirements, where functional requirements specify what the developed software should do e.g. data, function and interfaces and non-functional requirements specify under which constraints the functional requirement should be fulfilled. The most important requirements are mostly non-functional requirements, such as performance, quality and requirements for the methods of realization, implementation requirements and organization requirements.
 - For the development of ISS, the emphasis will be put on the dependability (safety) related requirements, which are mostly non-functional requirements. Here the relevant dependencies between the system-components, the risk mitigation requirements should be included in the dependability related requirements, e.g. switch-off of ACC in case of sensor failure of ESP.

The result of this process step is an improved, prioritized and categorized requirement specification, which is the most important milestone and basis for the requirement engineering work. As a living document, the specification should be supplemented and traced during the whole ISS Engineering Process.

As proposed, the tools DOORS and SysML will be used. One trend which should be mentioned here is the so-called model based requirement engineering, in which the requirement specifications are mapped to the model based functional design. With the tools of DOORS and SysML, the structured requirement specification can be further linked and mapped to the appropriate UML-design or model-blocks with Matlab/Simulink. Further on, they can be linked to the V&V requirements and approaches,

Process step 2.1: Specify functional architecture

Specify functional architecture is a typical step of "divide and conquer". The whole system is decomposed according to domain-specific considerations. While ISS-applications are characterized with inter-domain and inter-ECU distribution, the developer should not lose the system overview and the interaction between the sub-components.

Process step 2.2: Specify dynamic behavior

Here a sequence of functions is defined. Eventually these must conform to the functional architecture defined in step 2.1. By using UML2.0 or SysML, e.g. the standardized notation of interface definition, the time sequence editors and use-case diagrams, the interaction between the sub-systems can be defined.

Process step 2.3: Specify Function Behavior

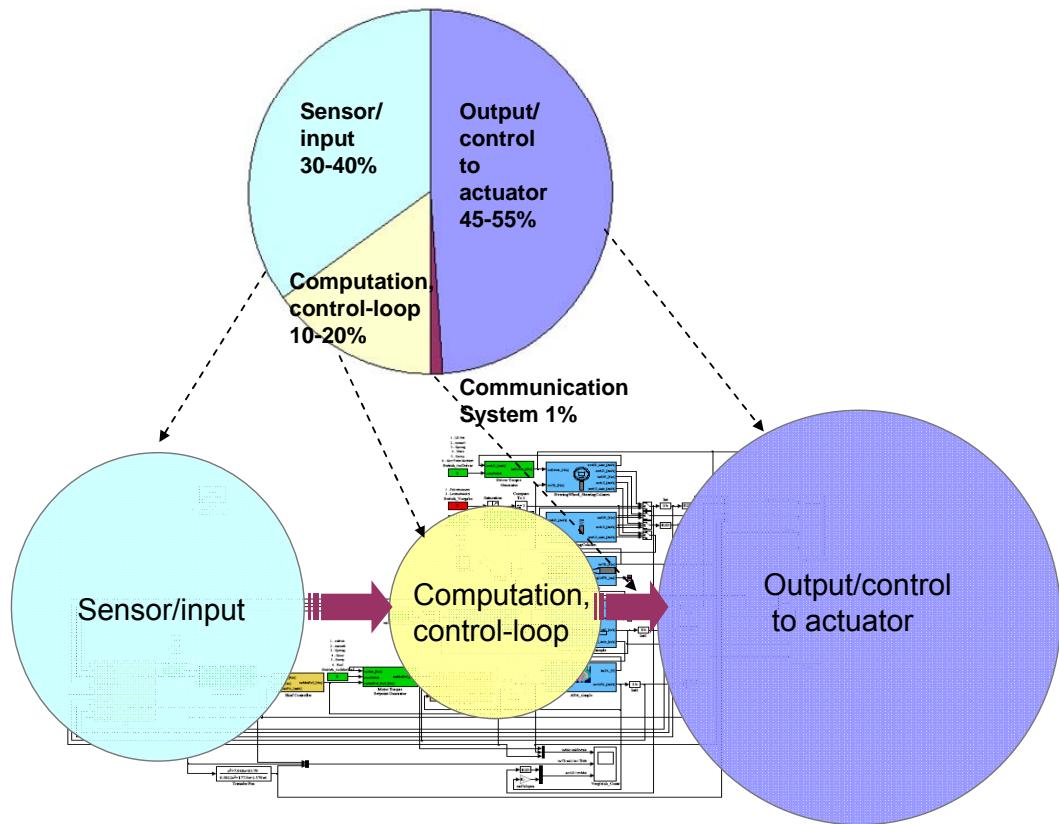
While steps 2.1 and 2.2 mainly concentrate on the "outer" view of a functional architecture, this step emphasizes the "inner" view as it cares for the implementation of the fine-grained functional behavior. The development of embedded software by means of a model-based design is already practiced in certain organizations in the automotive and other industrial domains. However, it is still not state-of-practice, esp. for the safety relevant systems. With the introduction of improved model based development tools, certified code generator and compiler, it can be foreseen that not only in the rapid prototyping phase but also in the serial development of safety relevant systems, the model based approach will be applied.

Process step 2.4: FAA Hazard Analysis (FHA)

FAA Hazard Analysis (FHA) extends and updates the results of PHA, through detailed analysis of constituent subsystems and components of the vehicular system, to identify previously unidentified hazards and to assess the associated risks.

Measures to perform the process step:

- Identify hazards and analyze their effects
- Identification of faults that can lead to the hazards, updates the hazard list and fault types/hypothesis of the system
- Identify risk mitigation measures and design criteria
- Allocate Automotive Safety Integrity Level (ASIL) to the FAA components according to ISO26262-3 - concept phase [ISO23]. With a qualitative analysis the safety integrity requirement (quantitative requirement of PFH) can be mapped to FAA-model. As shown in Figure 7-7, based on experience, the system PFH can be mapped to the FAA-components of sensor/input data processing, communication, ECU computation and output to actuator.



- Output/control to actuator: 49%
- Computation and control-loop: 15%
- Communication system: 1%
- Sensor/input: 35%

Figure 7-7: Mapping of system-PFH to FAA-components

A plausible mapping example is shown as above with ASIL-D system PFH as $1*10^{-8}$, the PFH of the communication system should not exceed the $1%*1*10^{-8}$ as $1*10^{-10}$. It demonstrates on one side the possible weakest chain in the system, such as communication system. On the other side, in order to improve the system PFH, one should concentrate on the system components as sensor and actuator control.

Results of the process step:

- A supplemented list of hazards (with additional and refined hazards compared to PHA) and mapping to the defined dependability specification in A4.3.
- Assessment and refinement of severity and risk level for each hazard
- Residual risk level of each identified hazard
- Additional and refined requirements for residual risk mitigation
- Additional and refined requirements for V&V, an initial mapping here should be performed to the test specification of the identified hazards. IEC 61508-3 calls for a test specification as the result of the functional step “Module Design” (ISS Engineering Process part 4). But already in this very early phase, the software developers, who are responsible for producing the test specification, and in particular the design of test cases, should make thoughts of test specification and their mapping to the requirement specification [SMT06].

Process step 2.5: Validate FAA model

In this step an early validation of FAA model (a typical step of virtual front-loading) is carried out, in which the functional behavior of one or more functional units must have been specified. For the behavior description, an executable specification is given for each unit. The level of details in these specifications might still vary, ranging from an abstract and coarse grain specification (e.g. a simplified state-chart model) or a detail specification (i.e. definition) of the functional behavior. In either case it is assumed that a commonly used modeling language with executable semantics is used for the specification of the functional behavior, e.g. MATLAB/Simulink/Stateflow. This FAA model is still regarded to be independent from an execution platform, i.e. it does not contain mapping information resulting from mapping the model onto a specific aimed hardware platform.

In the validation process step a preliminary “model-in-the-loop” test can be carried out. It is checked whether the FAA model conforms to the structured system requirements specification. Therefore, a test model has to be derived from the requirements specification and fault types/hypothesis. The test model might be specified in the same language as the functional behavior. In this case the FAA model may be embedded in a test frame that has been triggered in a test environment by the test model, e.g. by performing an offline simulation in MATLAB/Simulink/Stateflow.

If the FAA model is given as a state-chart, formal verification techniques can also be employed. In this case the test model contains assertions and the verification which are used to check whether the FAA model fulfills these assertions.

If the FAA model reflects a control loop, also methods from control theory for stability analysis can be used. In case of a linear system there are standard methods available. To some extent there are also methods for non-linear systems, although these require additional effort.

The proposed tools for the validation of FAA models are DOORS (for specification of the system requirements), DOORS, MATLAB/Simulink/Stateflow (for the test model), test automation tools on the PC-platform (for the simulation engine and execution of tests) and dSpace Embedded Validator (for formal verification).

As depicted in Figure 7-1, there are two important mappings, the mapping of Functional Safety Concept (artifact A0) to the improved Requirement Specifications of A4 and the mapping of Test Specification of A4* to the Requirement Specifications of A4. Since the FAA-model developed in Part2 only contains the most elementary functionalities of ISS, the mapping here is only a first step for a complete Functional Safety Concept required in [ISO23], which is living-document through out the safety life-cycle. The technical safety concept here together with the Test Specification and test results form the basic parts of safety cases, which serve as the proof of the safety integrity level.

7.2.3 Part 3: Development of hardware architecture

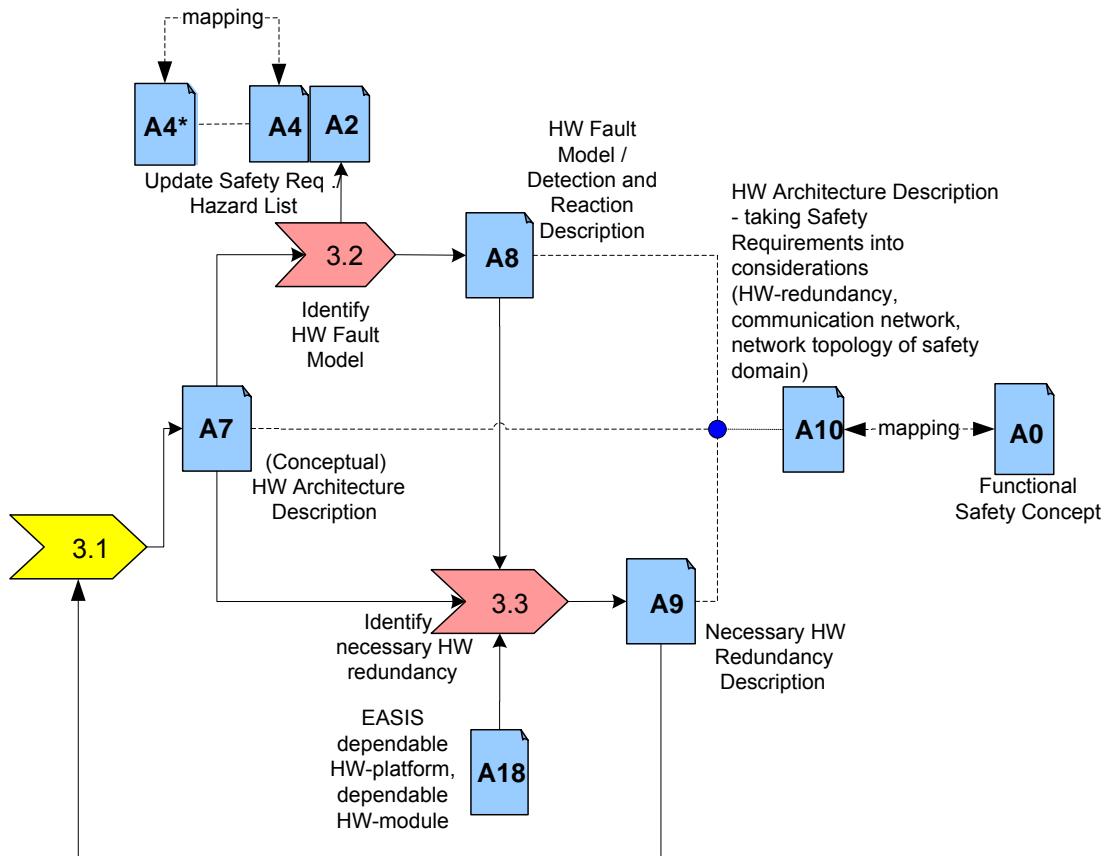


Figure 7-8: Development steps for hardware architecture

Figure 7-8 shows the consideration of dependability aspects by the development of HW Architecture Description (artifact A10). A system or hardware architecture represents one of several system design alternatives that are generated to explore the design space and to find an optimized solution. This optimum is mainly driven by the continuous system optimization, but for the safety relevant applications, however, the same safety case (the same safety integrity) should be provided despite of the optimization. Thus the hardware architecture should be mapped to the Functional Safety Concept as well.

Process step 3.1: Design system hardware architecture

As introduced in Subsection 3.3.3, the hardware architecture describes the vehicles topology, bus systems, ECUs, gateways as well as the employed sensors and actuators. The hardware architecture has to fulfill the system requirements specification, which already contains requirements to eliminate or mitigate hazards identified during early hazard analysis on system level. The analysis functions (FAA components) are tentatively allocated to the hardware architecture. This includes the functional devices of the FAA model that are mapped to the hardware, guidelines for the definition of the hardware architecture and the distribution of the functionality. The result is a conceptual hardware architecture description that can be analyzed and iteratively refined until it complies with the safety requirements. Necessary hardware redundancy identified during further design steps has to be incorporated into the hardware architecture. This means that either the description has to be updated or a new hardware architecture has to be chosen that is able to fulfill safety requirements but may not need the redundant parts. For example, instead of two separated redundant sensors, one sensor with integrated redundancies on one silicon-board with independent data paths could be selected.

Process step 3.2: Identify hardware fault model and risk mitigation measures

In order to identify possible hardware faults and to determine which faults may lead to which system level fault modes, a hazard occurrence analysis with e.g. a qualitative Fault Tree Analysis (FTA) has to be performed. The result of this activity is a starting point for the definition of error detection mechanisms and an action that has to be taken in case of a detected error. Within this step, since software related mechanisms are strongly connected with the fault-treatment, the software mechanisms here are covered in step 5.1. The proposed tools are tools for FMEA und FTA.

Process step 3.3: Identify necessary hardware redundancy

The application of dependable HW-architectures platform, e.g. redundancy concept with sensors has an impact on the hardware architecture as well as on the software design. The hardware architecture has to be iteratively refined for the redundancy. The hardware redundancy description establishes a specification to incorporate typical ISS dependability software services, such as voting and Agreement Protocol, into the functional architecture.

7.2.4 Part 4: Development of Functional Design Architecture (FDA)

- 1) As depicted in Figure 7-9, the Basic FDA Model (A11) is designed as the mapping of FAA to the hardware with the basis software services needed as OS, gateway-service and communication service, etc. No attention is paid here to the constraints of timing, bandwidth of communication network, etc.
- 2) With the new information of “basic FDA model”, the hardware fault model is supplemented to the system fault model based on the fault model defined here.

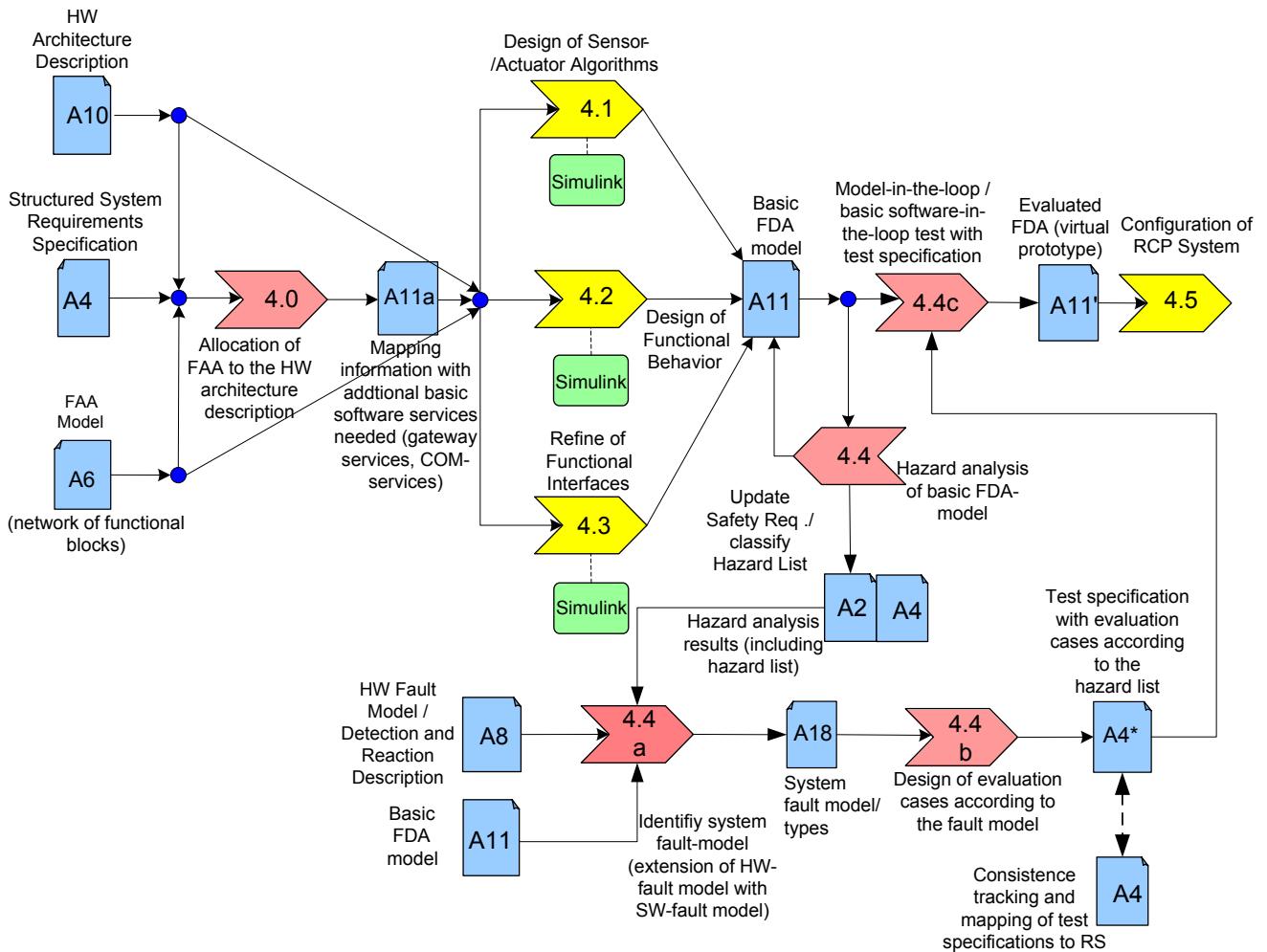


Figure 7-9: Design and validation of Functional Architecture Model with virtual front-loading

Here it is to note that one of the main differences between the FAA and the FDA is the level of details. The FAA typically does not consider domain-specific software infrastructure (e.g. diagnostics, state management and network management, etc.). The FDA with mapping on hardware, on the other hand, should consider these aspects.

Process step 4.0: Allocation of FAA to the HW-architecture description

In this step, a first version of the FDA-model of the function under development is generated. The basic idea of this process step is to combine the results of part 2 and part 3. The already existing FAA-model, as a functional network designed from model based development, is allocated to the HW-architecture description including safety requirements.

The model of application software (each can consist of more than one software component, e.g. sub-systems implemented with MATLAB/Simulink) will be mapped to the HW-platform (real ECU, µC or rapid prototyping hardware node), taking the system safety requirements and other constraints into consideration. That is to say, by the mapping of application software models to the rapid prototyping platform, esp. for the integration of ISS applications with different ASILs, safety requirements with regards to the reliability, bus load (communication effort) between the function

models, communication delay between the functions integrated on the nodes, etc. should also be considered for the first time here. As for some of the constraints, because no 100% accurate prognoses can be made, a rough estimation is enough to define the first FAA-model.

There can be more than one mapping possibility of FAA-model to the nodes of rapid-prototyping-platform. These mapping results should be assessed to the structured system requirements A4. Mapping of FAA to HW-architecture also implies a first rough decision for HW/SW-partitioning. While this is always a situation based case and there are no golden rules, a list of criterion is given as following:

- End-to-end timing: Hard real-time requirements can be often only implemented with hardware.
- Overall costs for the hardware and the software solution.
- Changeability during lifetime is easy to be implemented with a software solution. This is nearly impossible in a hardware implementation (except for FPGA).
- Know how protection: It is more difficult to perform reverse engineering of a hardware solution than of a software solution.
- Time for development: Algorithms can often easier and faster to implement in software than in hardware. The software based solutions are esp. preferred for prototype.
- Verified legacy solutions: If a solution is already implemented in hardware or in software, it is often beneficial to reuse it. However, even if a new solution would be more efficient, it is sometimes difficult to replace it, since the old solution is established.
- Background and competences of the developing team. In real life the project members have an influence on which selections are chosen for hardware/software partitioning.

Considering the correct-by-construction approach and reuse of prototyping here for the serial development, constraints of cost should also be considered. A guideline for the mapping work regarding the constraints is discussed in Subsection 9.3.2.

For the allocation of the FAA-model to the hardware architecture, there are in practice no mature tools for the design work. Vector DaVinci supports the mapping of FAA-model to the hardware architecture according to the principles of runnables defined in AUTOSAR as explained in Subsection 9.3.2.3.2, while the expertise of considering the multiple constraints and trade-off between them remains the responsibility of the developers. No intelligent and automatic mapping support can be foreseen in the near-future.

Process step 4.1: Design of sensor-, actuator-algorithms (I/O incl. bus I/O)

Based on the chosen hardware (mapping to the hardware), the algorithms for evaluating the respective sensors and actuators must be designed. The proposed tools are MATLAB/Simulink extended by Real-Time block-sets (dSPACE) with bus-interface and INTECRIOS including ASCET-SD (ETAS).

Process step 4.2: Design of functional behavior

While the FAA provides a coarse description of the functions, this process step is charged with the design of the algorithms that carry out the function. Moreover, additional issues such as calibration and mode management must be taken into account. The proposed tools here are MATLAB/Simulink and ASCET-SD.

Process step 4.3: Refinement of functional interfaces

Stemming from algorithm design, the requirements of the input values as to resolution, minimum/maximum and error values, etc. are known. The same is true for the values provided as output of the algorithm. Thus this process step is charged with the exact definition of the interfaces in the value domain and to the dynamic behavior - also the timing domain. The proposed tools here are TargetLink (dSPACE) and ASCET-SD.

Process step 4.4: Basic Design Hazard Analysis (BDHA)

Basic Design Hazard Analysis extends and updates the results of PHA and FAA Hazard Analysis, through detailed analysis of constituent subsystems and components to identify previously unidentified hazards and to assess the associated risks. The new components introduced by the work in steps 4.0 – 4.3 have to be analyzed.

Measures to perform the process step:

- Identify hazards and analyze their effects
- Identify risk mitigation measures and design criteria
- Assess residual risk level
- Allocate ASIL to the FDA components, in step 2.4 the allocated ASIL to FAA will be verified here with the additional mapping information to hardware.

Results of the process step:

- Enlarged list of hazards (with additional and refined hazards compared to PHA and FHA)
- Assessment and refinement of severity, probability and risk level for each hazard
- Residual risk level of each identified hazard
- Additional and refined requirements for residual risk mitigation
- Additional and refined requirements for V&V

Proposed tools are still tools for FMEA and FTA.

Process step 4.4a: Identify system fault-model

Step 4.4a is in fact a sub-step of step 4.4, with the complexity of ISS, the identification of system fault model is outlined. In this step, the existing hardware fault-model from step 3.2 will be reconsidered with the new knowledge from the FDA model. Generally speaking, the hardware faults are mostly isolated faults except for the common mode fault as the power supply, while for the ISS, the distribution and mapping of software to the hardware and communication between the

components, an integrated view of system wide fault model with software fault model as introduced in Chapter 6 should be considered here.

Process step 4.4b: Design of evaluation cases according to the fault model

The mapping of improved fault model/fault types to the evaluation case and from the design of evaluation cases to the test specification as well as front-loading of system test is a consistent evaluation process in the ISS Engineering Process.

Due to the complexity of ISS, the design of evaluation cases should follow the same principle of Fault Tree Analysis with the knowledge of FAA and FDA. Since the possibility of fault propagation is amplified in ISS, the evaluation cases with fault injection on the remote node should be considered here as an emphasis. One of the aims of designing such intelligent evaluation cases is to find out the side effects, which are not foreseen in the design phase.

The mapping between items in the fault model to the test specification and later evaluation cases is an n:m (n to m) mapping, which should be checked for the completeness. The tools suggested here are again SysML and DOORS, which enable the mapping of model based evaluation cases to the text based test specification.

Process step 4.4c: Model-in-the-loop/basic software-in-the-loop test with test specification

With the mapping information in the FDA-model, an initial separation of system model to the hardware platform can be made. The communication between the separated models (the initial nodes) can be emulated with the models of communication network using the appreciate block-sets provided by the simulation tools. The environment and driver input can be emulated now more realistically, since accuracy and hardware constraints like interface, specifications of input signals from sensors and feedback from actuators can be included in the models step-by-step. With evaluation cases by fault injection according to the fault model, not only the system functionality but also the system dependability behavior in presence of faults can be evaluated as well.

In addition to the model-in-the-loop test, instead of using the simulation tools to emulate some of the basic services in the models such as COM, TP-services or normal gateway services, these basic software services as well the applications above can be compiled together. Here it is notable that these general software services can be chosen independent from the aimed HW-platform or the specific hardware are physically available. Both Matlab/Simulink and Vector CANoe or DaVinci support the binding of compiled ECU-model e.g. Dynamic Link Library file (DLL-file) in the simulation model. Thus a more near-praxis simulation and front-loading of design evaluation of risk mitigation can be carried out.

Within the framework of model-based software development in conjunction with automatic code generators, the test cases designed for the MiL-test can be also reused for the SiL test.

Process step 4.5/4.6: Configuration and validation of RCP system

As shown in Figure 7-10, the basic FDA model will be configured for the rapid prototyping process. According to the certain tool chain and requirements from rapid prototyping hardware, prototyping

specific parameters, blocks configuration and environments will be integrated to the FDA-models. One example here is the configuration of FlexRay scheduling for Rapid Control Prototyping (RCP) using the DECOMSYS tool chain. The separation of communication mechanisms and scheduling of communication table using DECOMSYS tools require that certain guideline of designs should be followed.

In order to customize the FDA-model for the RCP, based on the validation results of RCP, certain iterations should be carried out.

7.2.5 Part 5: Refinement and validation of FDA model with SiL-test

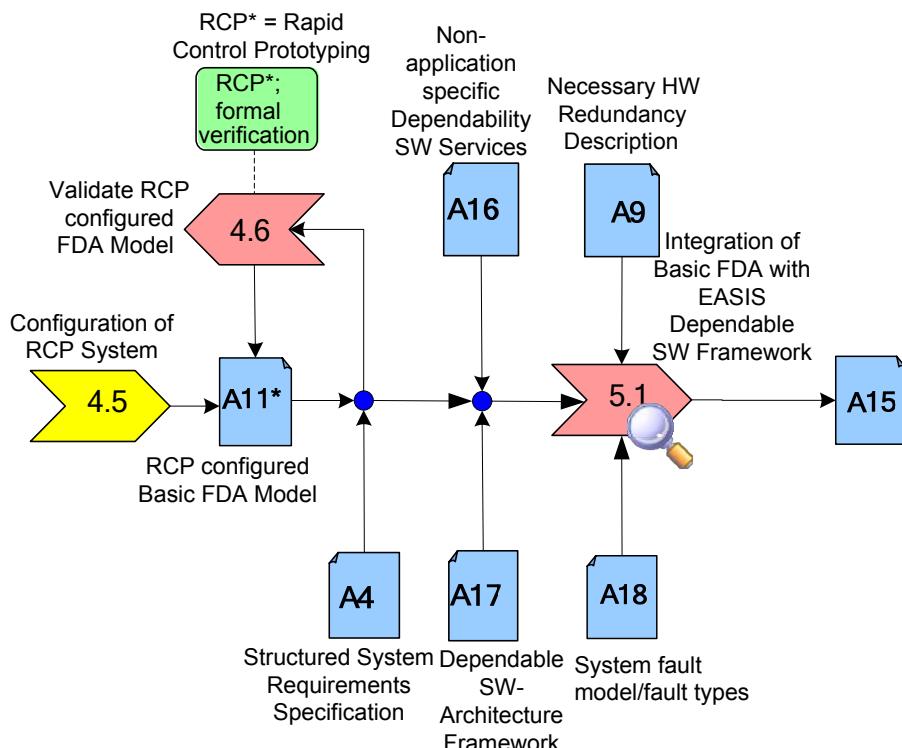


Figure 7-10: Refinement of FDA model with dependable SW-platform

Process step 5.1: Integration of Basic FDA with EASIS Dependable software Framework

In this step, based on the RCP configured basic FDA model as depicted in Figure 7-10, hardware redundancy description and hardware fault detection mechanisms, appropriate dependability software services from the dependable architecture framework as introduced in Chapter 9 are chosen to support the fault detection and fault treatment mechanisms.

The application software, dependability software services and the basic software services will be configured as specified in the dependable software architecture framework and integrated into a first complete FDA model.

As depicted in Figure 7-11, details in process steps 5.1.1, 5.1.2, 5.1.3 and 5.1.4 are defined as following.

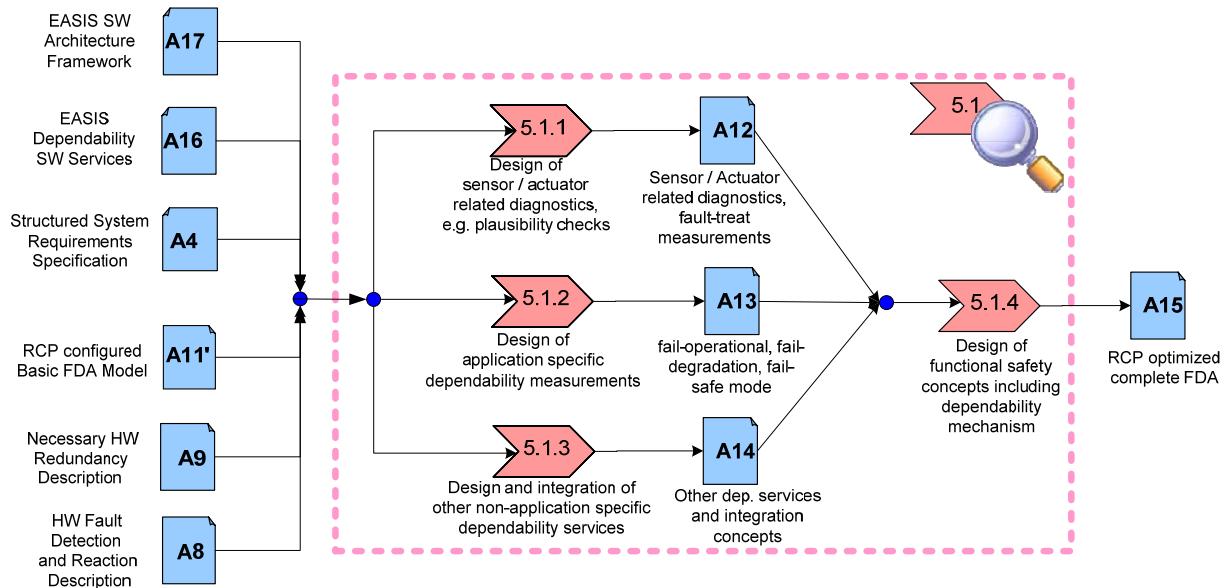


Figure 7-11: Process step 5.1 under magnifying glass

Process step 5.1.1: Design of sensor/actuator related Diagnostics

In this step, based on the specified functional system description and the characteristic behavior of the sensors/actuators in case of sensor/actuator faults, an appropriate diagnosis strategy is defined for them. In EEP step 4.1, the basic handling of (virtual) sensors has been designed. In step 5.1.1, the algorithms to detect the possible errors are integrated into the FDA model.

Process step 5.1.2: Design of application specific dependability measurements

The most plausibility checks are strongly connected with the design of applications. Based on the specific application requirements and the hardware redundancy, appropriate dependability software services will be configured to perform the plausibility check of the following entities:

- Sensor signals
- Received signals from communication system, esp. the plausibility of the signal combination, e.g. based on the relationship among temperature, pressure and volume of engine, some plausibility assertion of the sensor signals can be made.
- Intermediate calculation results
- Sent signals to the communication system and computed desired value to the actuator.

Process step 5.1.3: Design and integration of other non-application specific dependability services

As introduced in Subsection 3.4.2.2, most dependability SW-services introduced here are application independent and can be configured to the application safety requirements. Such services as Agreement Protocol, fault-tolerant inter-ECU communication, Software Watchdog and Fault Management Framework can be selected and configured for the specific needs of the application with a reference to the dependability SW-architecture framework. According to ISO

norm 26262 part 6 [ISO26], the application of dep. SW-architecture framework, reuse and configuration of dep. SW-services should be documented individually for each software component.

Process step 5.1.4: Design of functional safety concept including dependability mechanism

Based on the concepts in A12 (sensor/actuator related diagnostics, fault-treat measurements), A13 (different modes for fail-safe, fail-silent, fail-degraded and fail-operational, etc.) and A14 (other dependability software services and integration concepts), fault tolerance mechanism of ISS-applications are integrated in the FDA-model. The dependable software platform as well as the ISS-applications should be optimized here for the rapid prototyping in another iteration as well.

Process step 5.2: Configuration and code generation for software-in-the-loop test

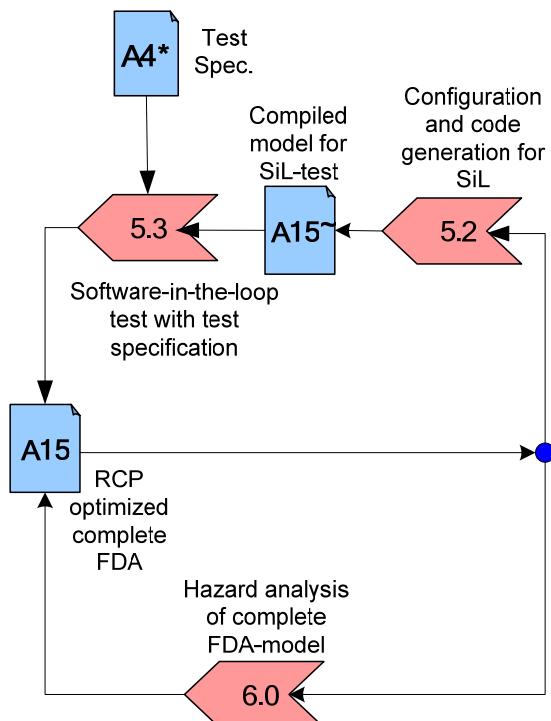


Figure 7-12: Validation of FDA-model with software-in-the-loop (SiL) test

In process step 4.4c, approaches of the model-in-the-loop-test and an initial software-in-the-loop test are integrated in the ISS Engineering Process. Compare with 4.4c, step 5.2 in Figure 7-12 is an extension with HW-specific features and system topology in FDA-model. In the software-in-the-loop test here, the generic hardware drivers and emulation of bus communication etc. can be substituted step-by-step with the µC or rapid-prototyping platform specific sensor/actuator related I/O-drivers, bus-drivers or other hardware libraries.

By applying appropriate code-generators and hardware specific compilers, the object codes can be generated and manual configured (e.g. additional check-point or break for the validation) for the aim of software-in-the-loop test.

Process step 5.3: Software-in-the-loop test with test specification

As mentioned in process step 4.4c, there are simulation tools supporting the embedding of compiled complete system model as a software based node. According to the availability of other nodes and developing phase, each node in the system topology can be chosen as the following seven variants:

1. Initial model, emulating very simple system behavior without any “intelligent” code behind.
2. Enhanced model with models computing system behavior based on the inputs without consideration of hardware constraints.
3. Software node with code generated from the enhanced model in variant 2, using generic hardware drivers, code generator and compilers.
4. Complete model as an extension of the model from variant 3, with integration of hardware specific issues like signals, timing and bus scheduling.
5. Software node with code generated from complete model in variant 4, using hardware specific drivers and code generators as well as compilers.
6. Hardware node with external bypassing, that is to say, the extended/new function is loaded to the rapid-prototyping platform, while the existing functions and I/Os are still integrated on the original ECU.
7. Hardware node, with complete code from variant 6 (bypassing and original ECU) loaded to the evaluation board or rapid-prototyping platform.

As depicted in Figure 7-13, in the different stages of development, from MiL, SiL to HiL, nodes are substituted successively with the advanced variants, while a consistent reuse of test cases and traced comparison of test results should be guaranteed.

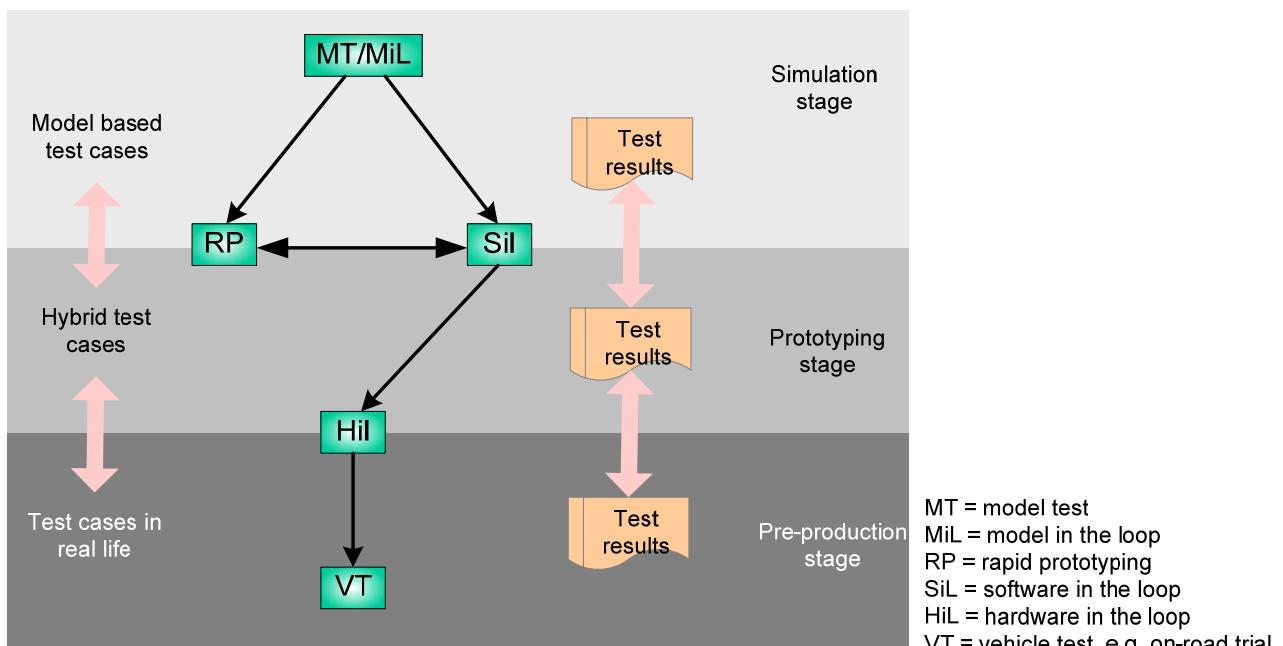


Figure 7-13: Test in different stages

A complete system topology is made up of nodes and communication system. While the nodes in the software-in-the-loop test environment can be one of the seven variants introduced here, the communication system can be also chosen from the following three variants:

1. Bus-communication connector emulated by the model-based development tools. In this case, no bus-scheduling, communication matrix or signal/frame lengths need to be considered. Most of this kind of information remains unknown in this early phase of development.
2. Emulated bus communication: Once the type of communication bus is chosen according to the bandwidth, safety requirement, complexity and cost factors, with help of rest-bus-simulation tools, the physical bus can be simulated with defined parameter as transfer speed, timing and frame length, etc.
3. Rest-bus communication: When there is the need to test real bus communication, the communication bus can be substituted piecewise with real communication bus. Once one of the node or some the nodes are available in the advanced format (from the initial model of variant 1 to the hardware nodes of variant 7), the nodes can be substituted one by one with the advanced form. Thus the whole software-in-the-loop test bench is a free-configurable hybrid system of SW-nodes and HW-nodes, which evolve towards the hardware-in-the-loop test bench.

The advantages of the rest-bus simulation are significant for the distributed development between OEMs and suppliers in both time domain and space domain.

- From the time domain: since traditional development cycle of a totally new modern vehicle model is about 3 to 5 years, while the development cycles of microcontroller are much shorter, most of the ECU or µC are not available or only available in form of evaluation board at the early phase of development of in-vehicle electronics.
- From the space domain: The development of in-vehicle electronics is geographically distributed as a cooperation of OEMs and suppliers. The front-loading of concept evaluation and integration test should be carried out almost parallel to the coding and implementation work on the side of suppliers. It is much simpler and quicker to transmit software models in form of a virtual node, for example, over networks than to ship evaluation boards.

In order to overcome the bottleneck from the time and space domain, the software-in-the-loop test with the rest-bus simulation provides the proper approach for the cooperation between OEMs and suppliers, while keeping the consistent test specification and requirement tracing with the later phase of development.

The common approach of the suppliers, to show the OEM the compliance of software quality assurance measures, is the software module test. In step 5.3 it is the first time that software codes are generated and implemented. According to IEC 61508-3, the software module test in form of SiL should be performed directly after software coding. This SiL-test provides the first opportunity to detect any deviations between the specifications and the software or to find any weak points in the software and to rectify the problems at an early stage.

7.2.6 Part 6: Hazard analysis and validation of FDA model with HiL-test

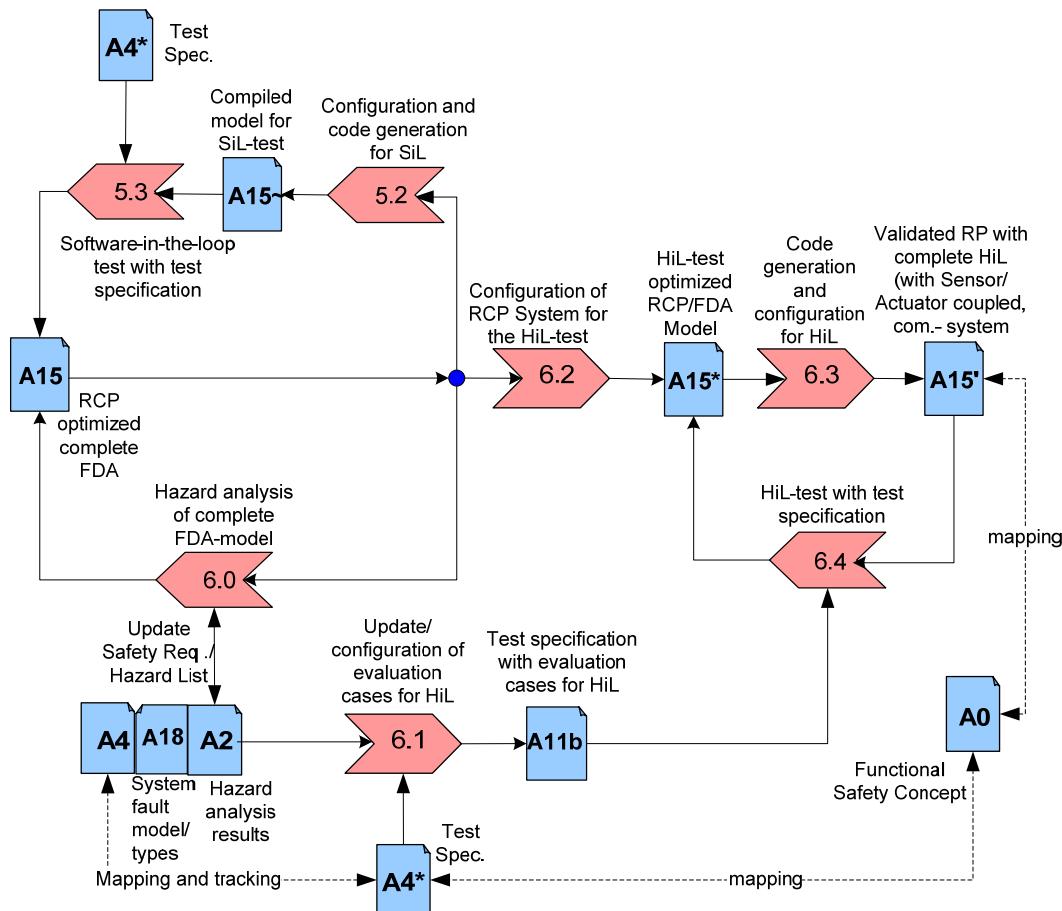


Figure 7-14: Validation of FDA-model with hardware-in-the-loop test

The term “hardware-in-the-loop” (HiL) test is widely used in the automotive industry. The hardware-in-the-loop test for the serial development usually refers to a large and sophisticated test bench with all the serial ECUs, sensors and actuators connected with the real communication bus for the preparation of in-vehicle test, in which the environment as on-road test is simulated as well.

The “hardware-in-the-loop” approach in Figure 7-14 refers to the test bench made up of rapid-prototyping platforms and evaluation boards for the evaluation of FDA models.

Process step 6.0: Hazard analysis of complete FDA-model

Hazard analysis of complete FDA-model extends and updates the results of PHA in step 1.2 and hazard analysis of FAA in step 2.4, through detailed analysis of constituent subsystems and components of the vehicular system, to identify previously unidentified hazards and to assess the associated risks. The new components introduced by the work in steps 5.x have to be analyzed. Results from step 6 work as an update of safety requirements in A4, A 18 and A2. An iteration of re-mapping and tracking between requirement specification and test specification should be performed as well.

Process step 6.1: Update and configuration of evaluation cases for HiL

The evaluation cases from model-in-the-loop test and software-in-the-loop test of legacy can be updated and reconfigured for the hardware-in-the-loop test. The benefits here are quite obvious, it prevents the wheels from being reinvented while keeping the dependability requirements traced as formal test. The evaluation results can be compared to gain more knowledge about the design.

Process step 6.2: Configuration of RCP system for the HiL-test

The RCP-system should be further configured to get ready for code generation of the hardware-in-the-loop test, for example, according to certain system requirement, communication scheduling might need to be separated from the model, some evaluation cases might be embedded in the system model for the sake of test automation and fault injection.

Process step 6.3: Code generation and configuration for HiL

With consideration of hardware resource, requirement of evaluation/test cases, interface of sensor/actuator/communication system, availability of hardware platform, cost and other constraints, different rapid-prototyping platform can be chosen. With appropriate code generator and tools, source code of the models as well as the evaluation case can be generated or implemented and compiled for the hardware-in-the-loop test.

Process step 6.4: HiL-test with test specification

As discussed in step 5.3, with the substitution of the nodes from pre-variant in the early development phase with the advanced variants, a seamless evolution of the software-in-the-loop test to the hardware-in-the-loop test can be made with consistent reusable test specification and test cases. In a complete HiL test-bench, the test environment is more realistic, the timing-constraints can be checked under real-time condition (comparison with the safety requirements, assumptions and previous testing results is possible) hardware faults according to the fault-types definition can be injected directly to the system. The results of the evaluation results from the HiL-test can be looped back to the FDA-model quickly.

For the ISS, as introduced in Process step 2.4: FAA Hazard Analysis (FHA), the allocated ASIL to the components as well as to the system should be validated in the HiL-test bench as well. The validation results from the iteration in A15 according to the test-specification A4* will be mapped and integrated to the Functional Safety Concept in A0.

7.2.7 Association of ISS Engineering Process with ISO26262

As introduced in Subsection 3.5.2.3, the ISO norm 26262 (draft version) is currently the most promising and future oriented safety norm in the automotive industry. Although the most important measures suggested in ISO 26262 are integrated implicitly in ISS Engineering Process here, it is necessary to map the safety steps from ISO 26262 to the ISS Engineering Process. Safety systems, developed by ISS Engineering Process should prove their conformance to the ISO26262 finally, to show the evidence that all reasonable safety objectives are sufficiently met.

As note in Subsection 3.5.2.3.3 the ISO 26262 is divided into eight parts, due to the limited focus of ISS Engineering Process (as shown in Figure 7-2), only the following parts of ISO 26262 are associated directly with ISS Engineering Process: Part 2 Management of functional safety [ISO22], Part 3 Concept Phase [ISO23], Part 4: Product development System [ISO24], Part 5: Product development: Hardware [ISO25], Part 6: Product development: Software [ISO26] and Part 8: Supporting processes [ISO28].

Association with ISO 26262 Part 2

As depicted in Figure 3-20, development reference phase 4.4 in ISO 26262 [ISO22] is the initialization of product development. The mapping of this part is listed in Table 7-1.

ISO WD 26262 part 2: Management of functional safety	ISS Engineering Process
Subsection 4.4.6 [ISO22], in the safety management during the development process, for all categories of safety relevant systems (ASIL A, B, C and D), the safety responsibilities of the persons, departments and organizations responsible for each phase during development should be defined.	In ISS Engineering Process notated in Figure 7-4, the dependability related process steps should be ensured to be carried out by a safety manager, who is appointed by the project manager to take the role for functional safety management tasks.

Table 7-1: Mapping of ISO 26262 Part 2 to ISS Engineering Process

Association with ISO 26262 Part 3

The ISO26262-3 - Concept Phase [ISO23] includes three parts: system definition, hazard analysis and risk evaluation and safety concept, which are mapped to ISS EP in Table 7-2.

ISO WD 26262 part 3: Concept phase	ISS Engineering Process
Chapter 5 Initiation of the safety lifecycle, during the system definition phase the safety lifecycle should be initiated.	Because ISS Engineering Process is firstly conceived for the design and prototyping of innovative ISS-applications, the life cycle here is generally considered as new development.
Chapter 6.4.7, a safety goal (different ASIL categories) should be assigned as an attribute to the system, sub-system and components in this process step.	The development phase of hazard analysis and risk evaluation is mapped to the ISS Engineering Process 1.2, 2.0, 2.4, 4.4 and 6.0, as a continuous assessment step of system hazards according to the latest knowledge about the system design. Because the detailed risk analysis and assessment is not the emphasis of this dissertation, the detailed activities specified in [ISO23] Chapter 6 are followed in ISS EP without adaptation.
Chapter 7 Functional Safety Concept	In the ISS Engineering Process Part1 and Part2, the Functional Safety Concept is mapped to different ISS-steps. In Part6, the validation step is mapped to the Functional Safety Concept as well.
Result from ISO26262 Part 3 (Subsection 7.2) is the functional safety concept such as architectural decisions and the concepts including the warning and degradation concept as well as the redundancy concept.	The design steps of fault detection and failure mitigation specified in [ISO23] are all considered in ISS Engineering Process 5.1, while the design of driver warning is not in the scope of this dissertation.
Subsection 7.4.3: Derivation of functional safety concept is the allocation of ASIL requirements (combination/decomposition) to the system/sub-system/components.	ISS Engineering Process Step 2.0, 2.4 and 4.4. Based on the new knowledge from the design step, the overall system safety requirement (ASIL) can be decomposed to the sub-system/components, thus the dependability related RS in A4.3, as a living document, should be continuously updated. This decomposition of ASIL to the components is one of most important criterion of verification and validation for HiL-test in step 6.4.

Table 7-2: Mapping of ISO 26262 Part 3 to ISS Engineering Process

Association with ISO 26262 Part 4

The ISO 26262-4 - Product Development System [ISO24] covers the reference phase “System Design” as shown in Figure 3-20, which is mapped in ISS EP in Table 7-3.

ISO WD 26262 part 4: Product development system	ISS Engineering Process
Chapter 6 System design, Subsection 6.3 The prerequisite of this part is the system requirements specification.	Product of ISS Engineering Process 2.0 as a artifact A4
6.4.1 Process requirements: review of safety concept, safety requirements specifications, design specification	A4, A4* and mapping between them. In each development phase of the FDA-model, the appropriate design specifications will be updated.
6.4.2 Requirements for architecture	Integration of dependability architecture - ISS EP step 3.3 and A18, ISS EP step 5.1 and A17.
6.4.3 Avoidance of systematic faults, for all ASILs, separation of safety related sub-systems and other sub-systems, avoid impacts of common cause failures.	Definition of system fault model and fault type in ISS EP step 3.2, 4.4a and artifact A18. Other non-functional RS as design guidelines.
6.4.3.6 Modular system design	Application of dependable HW/SW-platform
6.4.4 Methods for detection and control of random failures	Dependability services in hardware and software in ISS EP.
6.4.6 System design partitioning into hardware and software	Development approach from FAA to FDA by mapping of FAA to hardware in ISS EP.
6.4.7 Verification of system design in System design inspection, simulation, prototyping, safety analyses	Virtual front-loading with RP and FMEA in ISS EP.
6.4.8 System design documentation	UML/SysML based design, FAA and FDA in model-based approach.

Table 7-3: Mapping of ISO 26262 Part 4 to ISS Engineering Process

Association with ISO 26262 Part 5

The ISO 26262-5 - Product Development Hardware [ISO25] covers the reference phase 4.6 to 4.7 (depicted in Figure 3-20) of ISO reference model, which is mapped according to Table 7-4.

ISO WD 26262 part 5: Product development Hardware	ISS Engineering Process
4 Hardware – Requirements analysis	The hardware RS in ISS Engineering Process 2.0 as a artifact A4
5 Hardware architecture design and implemented measures for the control of random hardware failures during operation. 6 Quantitative Requirements for random hardware failures	ISS EP 3.2: Identify hardware fault model and risk mitigation measures for systems of ASILs, while the failure rate analysis of the hardware is not the focus here. The detailed activities specified in [ISO25] are followed without adaptation.
7 Measures for prevention and control of systematic hardware failures	Not covered by the ISS EP, such hardware failures can not be tested by means of RP.
8 Safety hardware integration and verification 9 Qualification of parts and components, e.g. compliance of electronic components with its definition of safety	Not covered by the ISS EP
10 Overall Requirements for software hardware interface: development of a SW/HW interface of a microcontroller included in a technical safety concept.	Partly covered by ISS EP, e.g., definition and verification of timing constraints

Table 7-4: Mapping of ISO 26262 Part 5 to ISS Engineering Process

Association with ISO 26262 Part 6

The ISO 26262-6 - Product Development: Software [ISO26] is a parallel activity to the hardware development in ISO reference model. The mapping of the activities there is specified in Table 7-5.

ISO WD 26262 part 6: Product development Software	ISS Engineering Process
5 Software safety requirements specification with suggested methods of natural language or computer-aided specification tools or semi-formal methods.	ISS EP step 1.1, 1.4, 2.0, 2.3, 2.5 and 4.6 applied to all ASILs
6 Software architecture and design	Applied to all ASILs in ISS EP.

ISO WD 26262 part 6: Product development Software	ISS Engineering Process
6.4.1/6.4.2 Choosing of design methods and tools for non-functional requirements to software	ISS EP tool chains, e.g. documentation and compute aided-designs
6.4.3/6.4.4/6.4.5/6.4.6 Hierarchic software architecture, non-functional requirements to software, guidelines for the modeling of software	ISS EP 5.1, A16 and A17 application of dependability layered SW-architecture, Modeling of FAA and FDA with UML
6.4.7/6.4.8 Allocation of safety integrity level to the SW-Cs and evidence of independence with ISO26262-8 Annex Concept of the independence of software components (draft)	Partly supported by ISS EP with the introduction of design step of fault model/fault region
6.4.9/6.4.10 Hazard analysis in software and measures for detection and handling of errors	ISS EP 4.4, 5.1, A16 and A17 and iteration of the steps
6.4.11-6.4.16 Individual specification and verification of the SW-C/SW-architecture, e.g. reuse or sourcing and configuration	ISS EP 5.1 (esp. 5.1.3), A16 and A17
7 Software implementation	Applied to all ASILs, only partly covered because ISS EP with focus on RP
8 Software unit test 9 Software integration test	Partly covered by ISS EP 4.4c, 6.4 and following iteration steps, not all the testing methods suggested in the norm are included
10 Software safety acceptance test 10.4.1 A software safety acceptance test shall be planned.	Covered by ISS EP completely.
10.4.2 Appropriate test methods for the software acceptance test shall be selected, e.g. interface, HiL, HiL within ECU network and test in the test vehicles	Partly covered by ISS EP until HiL-test with ECU network
10.4.4 Test specification with test cases, tests from the preceding test phases and their results can be reused.	ISS EP follows this guideline exactly, with the consistent test case of mapping and reuse
10.4.5 The software safety validation shall be performed on the target system.	Not within the scope of ISS EP

Table 7-5: Mapping of ISO 26262 Part 6 to ISS Engineering Process

Association with ISO 26262 Part 8

The ISO 26262-8 Supporting Processes [ISO28] describes a broad scope of topics. The relevant parts to the implementation ISS EP are listed as in Table 7-6:

ISO WD 26262 part 8: Supporting processes	ISS Engineering Process
4 Interfaces within distributed developments	ISS EP itself covers more technical issues than the organizational topics; for discussions here related to ISO 26262, see Subsection 7.4.
5 Overall management of safety requirements such as communication, traceability (wherfrom, whereto), documentation and administration	Traceability and documentation are completely covered in ISS EP, while the other topics are discussed in Subsection 7.4.

Table 7-6: Mapping of ISO 26262 Part 8 to ISS Engineering Process

7.3 Tool chains for the development of ISS

A suggestion of tool chain to support the development of ISS is given here, in which the tool chain is categorized into software tools and hardware prototyping platforms.

Main categories of software tools:

- Tools to support requirements engineering process and requirement tracing as well as design process.
- Tools for development environment of embedded software, model-based tools, development environment and code generators.
- Tools to support interchange format global process, e.g. interface description and communication scheduling.
- Tools to support mapping processes of FAA to FDA.
- Scheduling design of communication system, synchronization of tasks and communication.
- Tools for test automation and design of evaluation cases with fault injection in the SiL and HiL test bench.

Main categories of hardware prototyping platforms:

- Rapid-prototyping platform of general purpose and Rapid-prototyping platform for special purpose with specific hardware interfaces.
- Evaluation board from semiconductor producers.

7.3.1 Software tools

- Tools for requirements engineering as listed in Table 7-7 [EAD42]
 - Gathering, categorization and documentation of the requirements
 - Requirement tracing in design, implementation and test phase
 - Change management of requirements to the design

Tool	Vendor	Features
Word, Excel	Microsoft	Capturing and structuring of requirements
DOORS	Telelogic	Support for requirements tracing
Requisite Pro	Rational	Full featured support for requirements engineering
CaliberRM	Borland	Full featured support for requirements engineering
IRqA	TCPSI	Full featured support for requirements engineering
ARTiSAN Studio	ARTiSAN Software	Requirements diagram with SysML

Table 7-7: Tools for requirements capture and analysis

- System Analysis and Design
 - Identification of functions (in terms of their interfaces) as part of an overall functional feature network (equipments and features of the E/E-architecture in plan).
 - Definition of the vehicle E/E-architecture (i.e. number and types ECUs and connecting bus systems including gateways).
 - Mapping of functions to ECUs.
 - Identification of the communication matrix (transmission of inter-ECU signals via bus systems, mapping of signals to frames, mapping of frames to busses, assignment of message priorities in case of event-triggered protocol, temporal scheduling of frames in case of a time-triggered protocol like FlexRay).
 - System level diagnosis, i.e. backward and forward error analysis.
 - System level analysis, e.g. feasibility of the distribution or allocation is checked, i.e., whether the requirements of the application system are satisfied, e.g., memory and processor capacity on the ECUs, capacity on the communication busses, and overall timing requirements.
 - Documentation of system design, test specification and verification results. By application of the software tools, while following certain development guidelines, a framework of such documentation can be generated automatically. Of course these documents should be reviewed and updated manually finally.

The software tools here are listed in Table 7-8.

Tool	Vendor	Features
DaVinci	Vector Informatik	Widely featured system level design for automotive software systems, software integration on single ECUs, embedded software design according to AUTOSAR principle
Preevision	Aquintos	Design and evaluation of in-vehicle E/E-architectures
INTECARIO	ETAS	Structured design of software components
DesignerPro	DECOMSYS	System level design with special focus on FlexRay as a time-triggered bus system
RT-Builder	TNI Software	System level design with special focus on real-time analysis
ARTiSAN Studio	ARTiSAN Software	Structured design of software components
Enterprise Architect ²	Sparx Systems	Structured design of software components with UML2.0

Table 7-8: Tools for systems analysis and design

- Function Analysis and Design

The tools listed in Table 7-9 are operated in a pure PC based environment. When the correct interaction with a real-time environment shall be validated, the functional models can be extended to operate on real signals of this environment.

Tool	Vendor	Features
MATLAB, Simulink, Stateflow	Mathworks	Graphical modeling and simulation of data and control flow models, code generation for prototyping purpose
ASCET	ETAS	Graphical programming language for data and control flow models
SCADE	Esterel Technologies	Graphical modeling of synchronous control systems
INTECARIO	ETAS	Rapid prototyping system supporting different models of computation
Real-Time Interface	dSPACE	Block-set extension for MATLAB/Simulink to support rapid control prototyping
MTest	dSPACE	Model based testing of MATLAB/Simulink models

Table 7-9: Function analysis and design

² For reasons of simplicity only one representative for the large family of UML tools is mentioned here. These tools are similar in scope and usually support UML 2.0 or derived UML profiles as SysML.

- Software Analysis and Design

The function design tools in Table 7-10 are usually accompanied by code generators, which convert the abstract functional description carried out by means of the design tool to source code. The latter can be further processed and compiled to ECU-ready executable code. The usage of automatic code generators can further contribute to reliability and safety aspects of ECU software. Also the manual coding of the control software, supporting analysis tools e.g. for code coverage analysis or code style checking can also be used in combination with code generators.

Tool	Vendor	Features
TargetLink	dSPACE	Production code generation, code coverage analysis and target simulation
Real-time Workshop	dSPACE	Code generation for rapid prototyping platforms
Embedded Coder	Mathworks	Production code generation
ASCET	ETAS	Production code generation
CodeWarrior	Motorola	Development environment for the Freescale microcontroller platform
Visual Studio	Microsoft	Development environment of C
Embedded Validator	OSC	Formal model checking
Polyspace Verifier	Polyspace	Code analysis

Table 7-10: Software analysis and design

- Tools in Table 7-11 are applied for the scheduling design of communication system, synchronization of tasks and communication in the development of ISS.

Tool	Vendor	Features
DesignerPro and SIMCOM	DCOMSYS	Separation of functional design and communication design, scheduling and configuration of FlexRay [DSC05]
CANoe FlexRay extension	Vector Informatik	scheduling and configuration of FlexRay, rest-bus simulation of FlexRay, gateway design

Table 7-11: Design of communication system with FlexRay

- Tools for test automation and design of evaluation cases with fault injection in the SiL and HiL test bench are listed in Table 7-12.

Tool	Vendor	Features
ControlDesk	dSpace	Central module of dSpace experiment software to manage and instrument the experiments with integrated Simulink interface for offline management of controller models and extended options for automation.
ControlDesk failure simulation	dSpace	A software component of ControlDesk to drive electrical failure simulation in the ECU cable harness, Remote control of the failure insertion unit.
AutomationDesk	dSpace	Environment for test automation of calibration and measurement tools as well as automatic test generation.
CANoe	Vector Informatik	Rest-bus simulation with SiL-test

Table 7-12: Design of evaluation and test automation

After this brief list of software tools, another important point here worth mentioning is the interaction between tools. Different tools offered by a single vendor are usually designed in such a way that they can be combined and interact with each other to solve complex engineering tasks as an integrated process. Beyond that, long standing partnerships between different companies have been established in the past, which enables the interoperability of product offerings, e.g. the interconnection between Matlab/dSpace tool chain and DECOMSYS tool chain.

In recent years research consortia and industry partnership have contributed to this by defining standard to enable the interoperability of tools. Among these groups are ASAM, HIS and AUTOSAR. Interoperability of tools can be accomplished by diverse technical means, such as (i) run-time interaction of tools via well-defined APIs and (ii) definition of interchange formats to be read/written by tools in the global process, e.g. interface description, communication scheduling with common communication design framework with XML.

7.3.2 Hardware prototyping platforms

There are wide varieties of hardware prototyping platform in the automotive industry. Here, a short overview about typical applied products is given:

- dSpace products
 - MicroAutoBox hardware: real-time fast function prototyping platform with a wide range of applications with CAN, LIN, and FlexRay interfaces.
 - Simulator hardware: Hardware-in-the-loop simulators and simulator-specific hardware for ECU testing.
 - AutoBox and modular hardware: AUTOBOX is an expansion box to load modular hardware with wide range of I/O boards and processor boards.
- Vector Informatik and TZM products
 - Network interfaces to CAN, LIN, FlexRay and MOST
- DECOMSYS products
 - DECOMSYS::BUSDOCTOR is a FlexRay monitoring node with measurement features connected to the FlexRay bus and visualization of bus communication.
- Mathworks products
 - xPC system: real-time rapid prototyping and hardware-in-the-loop simulation using PC hardware
- Evaluation boards from various semiconductor producers
 - Usually years before the serial microcontroller are available as mass product for the automotive serial production, the semiconductor producers will provide evaluation boards for OEMs and suppliers to test the main functionalities in the main-core and interfaces of the future products. The experiences gathered with the prototyping can be applied to the serial development directly.

7.4 The ISS Engineering Process under challenges

7.4.1 View from side of OEM

Similar to the development of in-vehicle electronics from other vehicle domains, the development of ISS is typically distributed between several partners.

Before the project initialization phase, the OEM will usually have an assessment of suppliers. According to ISO26262 [ISO28] 4.4.2, the key criteria here is the capability and experience of the supplier in the development of systems with comparable ASIL. In the initiation phase, persons in charge of safety issues (Safety Manager) should be assigned both for the OEM and suppliers. During the whole product life cycle, OEM will require a consistent practice of the ISO26262 from suppliers to their sub-suppliers as well as sub-contractors.

The OEM is usually responsible for the overall functionality of the vehicle, i.e. the OEM defines which features and sub functions are installed in a vehicle platform or in a specific variant. Looking at certification and homologation, the OEM is also responsible for the safety of the vehicle as a whole and must be able to prove that the vehicle is “safe enough” to drive on the road. Thus OEMs

should have enough skills in system integration, architectures of embedded systems and making strategic decision of employing new approaches for embedded systems.

Usually, the OEM does not develop all the system functions alone, but with the help of suppliers. As shown in Figure 7-15, the OEM defines the components and the interfaces that he then gives to suppliers for implementation. Another possibility to distribute workload is the splitting of work at certain steps. E.g. it is possible that a vehicle function is developed on FAA level by the OEM. The following steps are given to other parties for detailed development or for target integration.

The relationship between OEM and first-tier suppliers can be extended as a template for the relationship between a 1.tier supplier and his (sub) suppliers, and so on.

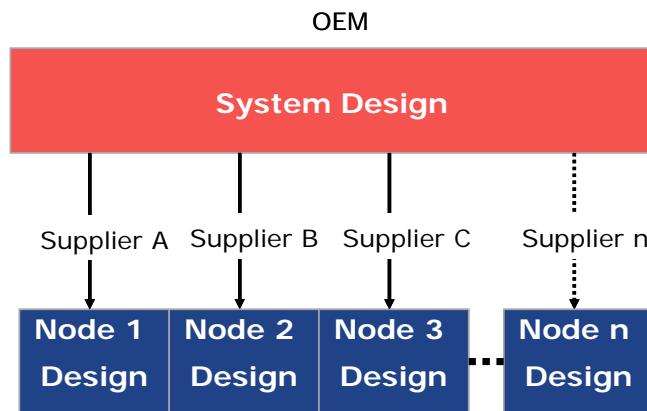


Figure 7-15: OEM - Supplier Workflow

With the challenges mentioned in the Subsection 3.5.3, the development of an inter-domain and inter-function ISS-application is characterized by an intensified distributed collaboration between OEMs and suppliers.

By the development of ISS, the OEM should cooperate much more closely with suppliers. OEMs (mostly high-end car-makers) add value by the integrating of different products from suppliers, thus the OEM should work as the initiator for the system architecture.

As depicted in Figure 7-16, in the first step, driven by the customer perceivable innovations, the features of the to-be-designed vehicle platform (as a basis of different vehicle model variants) are designed. The features are grouped to systems and prioritized to focus the efforts.

In the second step, a feature and function network (FDA in ISS Engineering Process) is defined, which is mapped to the third step hardware architecture to form the Functional Design Architecture. The ISS Engineering Process mainly covers from step 1 to step 3. The mapping process, disassembly and assembly of grouped and prioritized features to the functions in FAA, from FAA to FDA is considered as the enabling transmission between the steps, where the dependability aspects are considered with the ISS Engineering Process.

In the fourth step constraints, such as installation space in vehicles, cable harness (length, EMC), will be considered for the installation in vehicles.

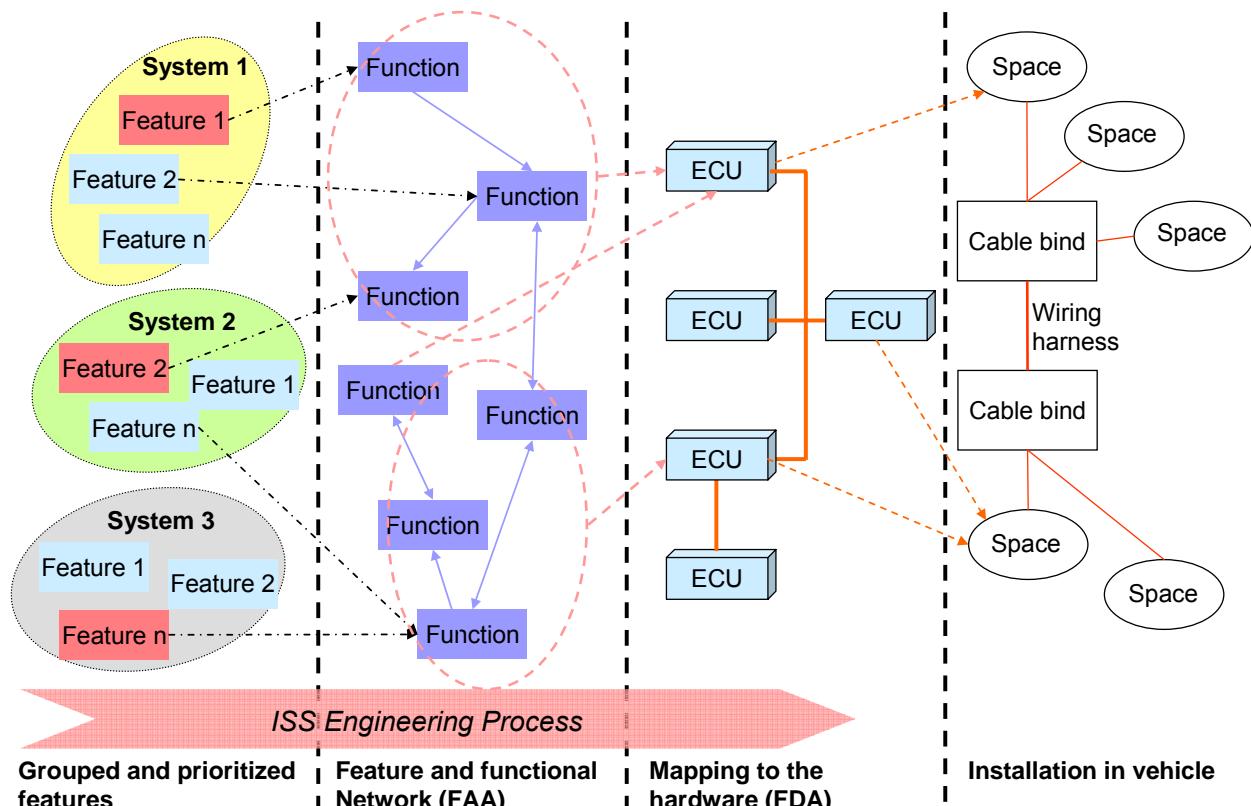


Figure 7-16: OEM view in the design of electric/electronic architecture

If the OEMs are only a “lean” branded integrator, the system suppliers should drive the architecture and provide integrated solution of the final vehicle with system know-how. In this case, however, an obvious disadvantage can be foreseen. The automotive OEM has to expose his vehicle function knowledge to the system supplier, who also supplies to the manufacturer’s competitors. It is quite challenging to protect intellectual property in such an environment. Re-implementation of vehicle functions results in design cycles of several weeks (outsourced engineering). Often, the function that might return to the system integrator does not fully match the required behavior, resulting in additional iterations because of misunderstandings.

7.4.2 View from side of supplier

On the suppliers’ side, a common approach is to gather the different requirement specifications from different OEMs to the comparable systems. At the beginning, the requirements from OEMs are gathered and merged together. By applying the design to cost principle, the requirements are sorted and prioritized as a whole.

In the development of ISS, the system supplier (first-tier), is responsible for the development of the system (more than one components) and must be able prove that the system meets all the requirements incl. safety requirements.

Considering the sophisticated system design, it is reasonable to build up one lean system architecture to integrate the most important functionalities, using standard dependability software architecture and standard hardware modules. In this early phase, the majority of effort can be focused on the design and validation of this lean prototypic platform instead of provision of

customer specific solution. The approach of applying platform with standard software and hardware architecture enables an efficient and cost optimized configuration and extension to the additional requirements of each OEM.

Thus the “wrappers” and too much customization parallel to the end phase of finishing the validation of the lean-system can be reduced. Efforts of development teams can be transferred to configure the customer specific features and fine calibration.

Particularly for the development of safety relevant systems, confirmation measures to the safety goals according to ISO26262 [ISO28] 4.4.9 “safety assessment at supplier's premises” are more penetrative to the supplier. Except for the review of FAA, FDA as well as the safety concept suggested in ISS EP, supplier should provide “safety cases” as safety evidence of all ASILs. Additional for ASIL-C, FTA, safety assessments and safety audits should be executed by the supplier. For ASIL-D, common cause failures (CCF), common mode failures (CMF) should be considered as well.

7.4.3 Distributed cooperation between OEM and suppliers

In the following, the two most important enabling levers for improving the distributed development between OEM and suppliers are discussed:

Process:

By the development of ISS, the OEM should have a clear defined role and core competencies. The OEM should agree with the suppliers on the interface in the cooperation and responsibilities.

One good practice is that the OEM involves suppliers in the early phase of system design. OEM's define high level architecture and standardization guidelines, but leave enough freedom for the supplier to decide on the implementation approach as software libraries and modules for effective reuse. The design guide, design approach, implementation experience, lesson learned and test specification in the rapid-prototyping phase can be seamlessly transferred and reused in the serial development and production. In the later phase, a well structured development model is applied to let the supplier develop the specification further. The prioritized features and requirements are traced together, while the supplier focuses on the component test and the OEM takes the role of system integration and system dependability test. Here it is noteworthy that both OEM and supplier should agree on the integration effort provided by both sides.

Architecture

The application of standardized dependable software architecture in ISS creates benefits for a typical “win-win” business situation between OEMs and suppliers.

- OEM can share across better validated legacy platforms software with suppliers.
- The inter-domain interfaces in the ISS (e.g. engine – transmission) can be serviced better.
- The platform strategies create technical pre-conditions for business with “software options”
- It leaves room for improvements and innovation, because an ISS system based on platform provides more flexibility for adaptation.

8 Hardware architectures for the Integrated Safety System

In this chapter concepts about the hardware architecture for the Integrated Safety System are discussed as a framework from different aspects. Based on the ISS-requirements for the hardware architecture as introduced in Subsection 3.3.1, the most important building-blocks of the hardware architecture for ISS and its configurations are specified here. As introduced in Chapter 7 about ISS Engineering Process, after the system design along with the technical safety concept [ISO23] is created, the next step is the design of the hardware architecture according to the steps in ISS EP Part 3 (Subsection 7.2.3). The emphasis of this chapter is focused to the design of hardware architecture of highly safety relevant ASIL-C and ASIL-D applications (as defined in Subsection 3.5.2.3.4), while the hardware architecture for ASIL-A and ASIL-B is not considered in this chapter.

The structure of this chapter is organized as following:

As introduced in Subsection 3.3, the automotive hardware architecture refers here to the three main views: system topology, communication systems and ECU architecture. In the following subsections, the design of hardware architecture for ISS is discussed from these three aspects. The last subsection here gives to each aspect a design use-case (based on dummy ASIL-D ISS applications), which concludes the discussion as about the hardware architecture framework.

8.1 Concepts of the system topologies

Following the V-process and the discussion as shown in Figure 7-16, usually OEMs take the responsibility to define the vehicle wide system topology and communication system, while the system suppliers and sub-suppliers define the ECU local hardware architecture according to the requirements of OEMs.

Generally speaking the definition and assessment of system topology for the in-vehicle electronics is a sophisticated challenge, which is influenced by many constraints, like application real-time requirement, cost, packaging, dependability, communication bandwidth, reuse of legacy system (take-over parts), maintenance, platform strategy of different models, supplier strategy, etc. This list of various constraints can be extended even much longer and much more detailed. What makes it even complicated is the weight of each metric/constraint for each OEM is seldom the same. It is even different for different product lines within one OEM.

For the discussion in this dissertation, since the cost-analysis and packaging issues are only meaningful for the design of a certain vehicle product line and beyond the dependability discussion here, the focus was chosen on the design of ISS hardware architecture in the early phase according to functional and safety requirements of ISS-applications. The granularity was limited on the definition of system topology from a high layer.

The approach here can be started with the question of which ISS-applications/features do we need for a certain vehicle platform, which extensions are expected on this vehicle platform. The answer of this question can contain the following parts:

1. A list of prioritized features or applications
2. To realize these functions, a list of certain components is needed e.g. sensors, actuators and interfaces (also HMI)
3. Furthermore the applications can be broken up into functions, which are connected to each other via interface for information exchange. (a very first functional network on the application level)

While the answer to 1st and 3rd question is a rough FAA model (as shown in Figure 7-7), the answers to the 2nd question is like a collection of “Lego”-modules, directly resulted from the features. With the first answers to these questions, the design of system topologies can be started.

8.1.1 Future frameworks and design guidelines of system topologies

As shown in Subsection 3.3.2, the in-vehicle system topology is typically organized in different domains, while the number of domains is mainly driven by the introduction of new features, which can not be integrated into the traditional domains because of e.g. lack of communication bandwidth, quality of services or management of domain complexity. On the other hand for the cost reasons the legacy domains and communication systems are preferred for the reuse, but at one point the needs of innovation and technical complexity to keep the legacy system running may overwhelm the cost benefit of take-over.

One of the most important design criteria, regarding the choice of domains, is the technical requirement of functions, in which safety is one of most critical technical requirements. To give an example, for the ISS-applications distributed beyond the domain borders, in each domain at least one gateway is needed for the inter-domain communication. In order to fulfill the functional requirement e.g. QoS and safety requirements communication e.g. PFH, etc., one potential approach is to use one high-speed communication network and connect all ECUs together. But on the other side a high-speed communication link is more costly than a low speed link. Thus a hybrid system with domains connected with gateway might be preferred.

In consideration of the ISS-Applications:

- Small and mid size cars have today two vehicle domains, body and powertrain. These two domains will be equipped with more add-on telematics service and safety applications.
- Today's high-end vehicles have three or four vehicle domains: body, powertrain, telematics and an optional chassis-domain, while the future high end car can have five or six vehicle domains, body, powertrain, telematics, chassis and optional HMI, environment sensing and/or safety domain.

For the further discussion, three types of possible future architecture frameworks are given in the following as vertexes. The future platform, however, can be a mixture of them [HWA06].

Figure 8-1 shows the backbone architecture, as a possible system topology for the high-end luxury vehicle platform, where the time-triggered communication bus connects the traditional vehicle domains and the chassis domain. Although the backbone bus and gateways between traditional networks will introduce additional development effort, this future oriented hierarchical architecture provides more potential for the cross domain communication regarding bandwidth and service quality. If some future ISS-applications should require a fault-tolerant or fail-operational behavior

for the inter-domain communication, the backbone architecture with time-triggered communication bus could provide the best preliminary condition.

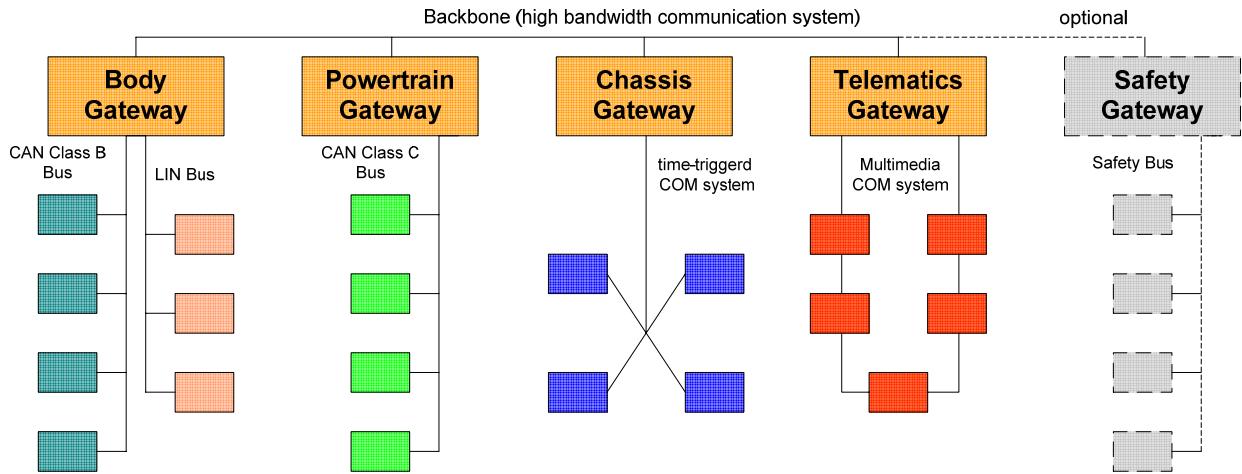


Figure 8-1: Backbone architecture

The other end of backbone architecture is depicted in Figure 8-2, in which the communication between the different sub-domains is provided by a single central gateway.

This solution requires a relative sophisticated gateway supporting a complex functionality to service all the different bus systems (e.g. bus interfaces) and to perform the routing and conversion tasks necessary. For the low-end vehicle, however, when not all the domains are required (in the simplest case only the CAN-B for the cabin/body domain and CAN-C for the chassis/powertrain domain), the central gateway architecture with simple CAN-routing is for sure a cost effective architecture solution.

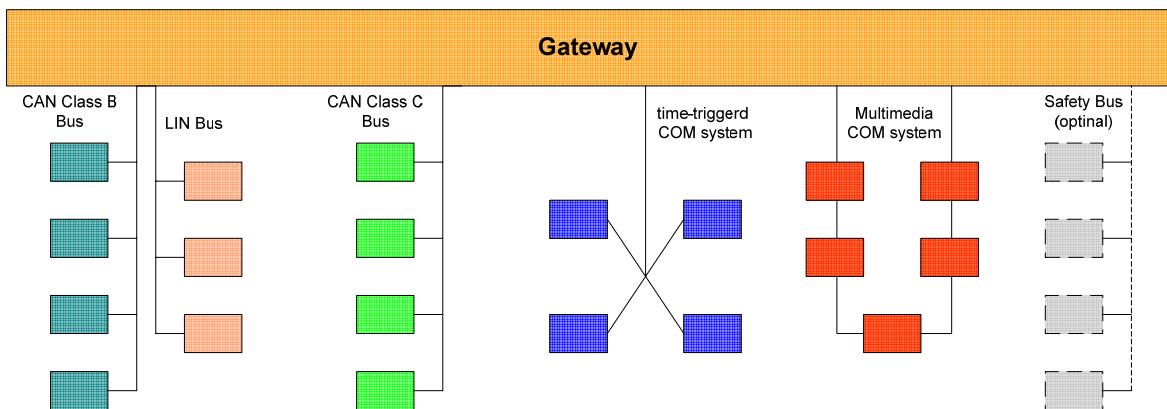


Figure 8-2: Central gateway architecture

A middle way solution is shown in Figure 8-3, where the sub-systems are provided by gateways integrated into single ECUs in a relative loose but flexible architecture. Low end and effective gateways (with 2 or maximal 3 bus-interfaces) are applied to connect the different domains.

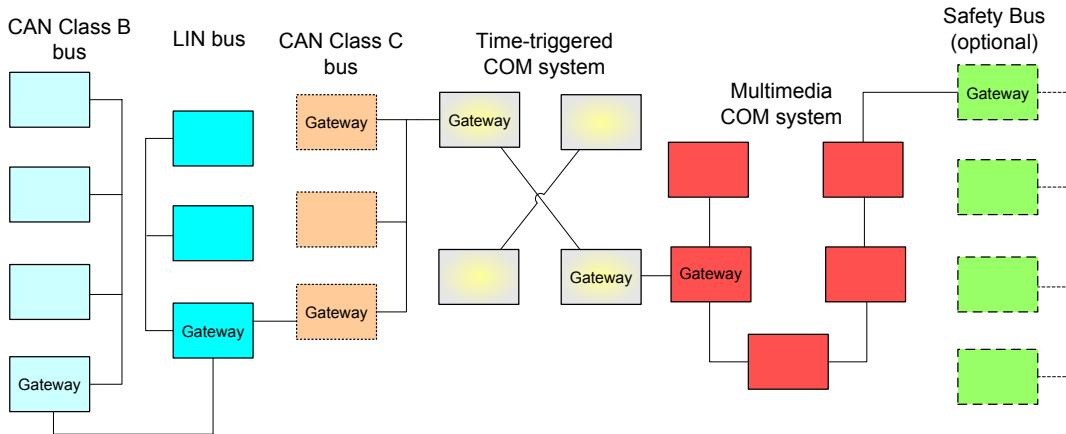


Figure 8-3: Multi gateway architecture

8.1.2 Distribution of ISS applications to the system topology

The FDA-model specified in ISS EP is derived from the mapping of FAA-model to the hardware system topology. The question how to distribute applications to the system topology is again highly application specific and there exist no general rules, taking all the constraints and guidelines into consideration. In the following, however, a few basic rules for the mapping and partitioning of ISS-applications are discussed:

1. Safety functions with hard real-time requirements to react to external events (e.g. tight control loops) must be placed on the ECUs, which are close to the event in the control loop, e.g. actuator control functions have to be placed on the ECU directly connected to the actuator.
2. Traditionally sensors are directly connected to the consumer of the information, usually to the same ECU as the actuator, thus the same vehicle dynamic data is redundant measured at different position of vehicles with appropriate resolutions and ranges for different applications. With the introduction of ISS, however, as long as timing (bandwidth) and safety requirement are fulfilled, the sensor can be displaced freely within the in-vehicle network (e.g. steering angle sensor). With distribution of sensor information, redundant sensors can be saved.
3. To keep the amount of data exchange, the network traffic under a reasonable limit, the applications, processing large blocks of raw data, such as data from camera or radar, should be mapped close to the source, e.g. by using intelligent sensors to provide information ready-to-use to the communication system.
4. Distribution of vehicle dynamic data with sufficiently high resolution (e.g. velocity, engine state, battery state and driver demand) can lead to new applications with sensor data fusion and share. But as soon as the safety relevant data is sent from one ECU to the other, the whole end-to-end data transmission path should inherit the highest safety integrity among the applications, which use this communication path. The application of

time-triggered communication is one possible solution but it alone does not solve the problem. Time triggered communication should be supported with other dependability software services for ISS such as application CRC and Agreement Protocol, etc. In this case event-triggered communication can be applied as well, if the communication can be supplemented with other dependability mechanisms as discussed in Subsection 8.2.

5. High level control functions can be usually placed on any ECU, where the necessary resources are available (processing power, memory and communication interfaces etc.), without degradation of performance. That is to say, the high level control functions could be freely shifted as long as software partitioning is guaranteed with e.g. the mechanisms discussed in Subsection 9.3.2.
6. Basic safety functions, such as brake, steering control and airbag, must not be integrated on the same ECU to avoid single point of failure. The approach of using fail-operational nodes without mechanical back-up in this case is still not suitable due to the system complexity in the near future.
7. For the same reasons, it is sometimes still necessary to keep the primary sensors of these basic functions redundantly distributed on different nodes. The distribution of sensor values via communication bus could make communication system as the weakest chain of the whole system, e.g. distribution of ASIL-D signals with CAN alone is not dependable enough for requirement of ASIL-D application. That is to say, if ASIL-D relevant signals should be transferred via CAN, additional safety services e.g. application CRC as discussed in Subsection 9.3.1 should be applied. In this case, the redundant sensors can be still applied, so that the ASIL-D application is able to work in the stand-alone mode in case of total communication failure.
8. For these redundant sensors mentioned in design rule 7, the state of the art situation is that little or less plausibility check among the sensors is applied. Such redundant signals can be used for plausibility check to gain more system safety, but the system availability might not be improved. Thus the local primary sensor should have a higher weight in the voting mechanism of plausibility check.
9. Reuse of legacy system (take-over parts) can be an important constraint by the distribution of ISS applications. Analysis shows that the warranty cost alone of ECUs (the additional development cost not considered at all) that have been significantly extended or redesigned compared to the predecessor model are up to five times that of reused ECUs [McK05]. Thus the cost/benefit-analysis of remapping should be carried out carefully.
10. If it is decided to integrate some of the safety functions on the same hardware, it is preferable to integrate only functions from the same domain (e.g. chassis-domain) with very tight communication/interaction on one ECU, because these functions usually have similar safety integrity requirements. On one side, due to the similar safety requirements, the same dependable hardware architecture can be applied. On the other side, the domain structure can be maintained, which is important for architecture simplicity.

11. The component safety integrity level can be reduced by intelligent mapping of safety functions of ISS to the hardware platform. This approach is illustrated with the following example:

The safety relevant driver assistant function Adaptive Cruise Control (ACC) can be designed that the ACC gives the ESP-system the braking command. If the braking command will be 100% executed by ESP, the ACC should be developed as the same ASIL-D application as ESP. Not only a 99% DC (Diagnostic Coverage) of ACC should be verified but also the signal path from ACC to ESP should be kept as redundant as well. But if we can develop a sub-function in ESP (according to the ASIL-C/ASIL-D requirements), which limits the brake command from ACC, so that even the ACC gives a faulty command, the vehicle can be still controlled by the driver and the following vehicles have enough reaction time, the safety requirement of ACC can be now reduced to ASIL-A. From this example, we can see, by intelligent design and distribution of safety relevant functions (mapping of the limiting function to ESP instead of ACC-ECU), the safety requirement to the component can be reduced.

12. For reasons of simplicity and reliability (e.g. signal latency), the communication path between two functions that need to exchange information should be kept as short as possible.

13. Distribution of redundancies of ISS-applications:

In order to meet the high dependability requirements of fail-safe and fail-operational, the conventional approach is to make use of redundancy. Typical examples of redundancies are separated power supplies, sensors, computation units, communication networks and actuators against common mode faults.

The integration of several functions on one ECU increases the potential threat of losing several safety functions at one time. Such threat can be reduced by the following two approaches:

- a. Integrate only “related” functions on one ECU, so in case of a failure only one branch of the safety function tree is lost. This approach is best used if it is combined with a smooth transition into the degraded mode, thus the driver has the chance to react to the degraded driving behavior of the car.
- b. Integrate functions on redundant ECUs, so that a single point of error does not lead to any degradation. Redundant units, can reside on the same hardware node or on remote nodes, which fulfill the same functionality in the system runtime or are activated when the original unit fails to work. By using dynamic reconfiguration (see Subsection 9.3.3.2), the backup unit can be “switched” on.

8.2 Design concepts of the communication systems

As demonstrated in the Subsection 8.1.1 about the different system topologies for ISS, in the near future there are basically two different in-vehicle communication systems, the event triggered communication bus such as CAN and time-triggered communication bus such as FlexRay. As mentioned in Subsection 3.3.4.2, although LIN and MOST can be applied in ISS, but they are not designed for the safety relevant applications, thus they are not discussed here.

CAN, as now the most widely used in-vehicle communication network, has proved itself as a cost effective and reliable bus system. The greatest constraint of CAN is the bandwidth and non-determinism since it was originally developed for the comfort application of cabin electronics and cyclic powertrain communication.

- As an event-triggered bus, CAN guarantees no worst-case transfer delay of messages, esp. in the case when a large burst of event-triggered messages are sent onto the bus. For the transfer of safety relevant messages like the battery status (voltage and key position), light controlling signal on Cabin-CAN, etc., dedicated measures have to be taken to secure the signal.
- CAN provides no time synchronization among the participants. A typical example problem is shown as Figure 8-4, when redundant steering angle sensors are connected with CAN to an ECU (consumer of the sensor signal). Since the redundant sensors can not be synchronized to each other to start the measurement at the same time and the transmission time over CAN could be slightly different, the result is that the measurements and the sending of the signal via the CAN to the respective sensor node are not synchronized but relocated with a certain time offset. Within this time offset the steering wheel can be turned with such a large velocity of angle, that the difference between two sensor signals overruns the safety relevant threshold.

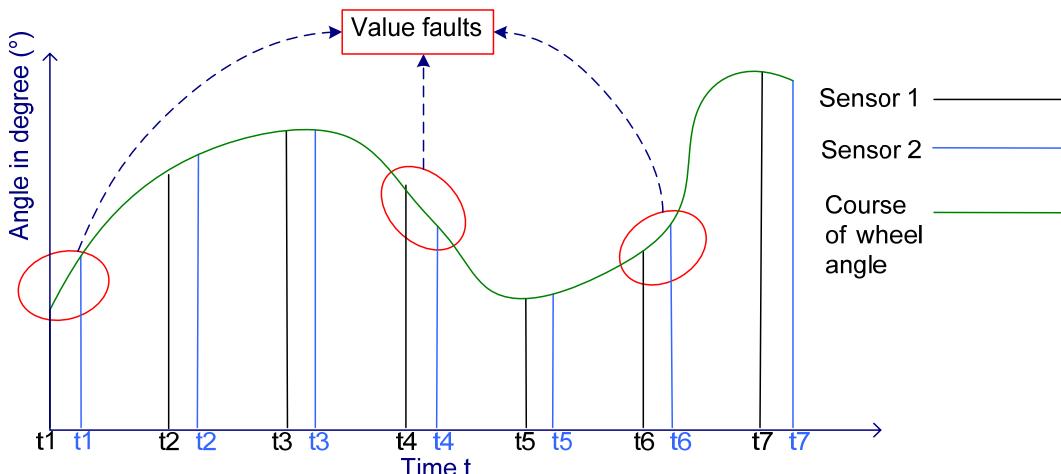


Figure 8-4: Different sampling of two sensors

- The limited bandwidth of 100 Kbit/s to 500 Kbit/s can be a bottleneck for future Integrated Safety Systems, which require deterministic communication with large volume of data exchanged between the nodes within the same domain or beyond the domain border.
- CAN provide only limited fault tolerant mechanisms such as CAN CRC-15.

Despite of these disadvantages, CAN is also applied for the safety relevant applications. The implementation of CAN for the specific application depends on the individual communication requirements regarding the quality of service. Generally speaking, there are following methods:

1. Using additional frame sequence number to monitor the transfer of CAN-messages, so that communication faults such as repeated or missing frames can be identified.
2. Using worst-case end-to-end transfer time (time-out monitoring) to monitor the message transfer on the receiver side.
3. Using acknowledgement to confirm the successfully transferred message by e.g. handshake protocol.
4. Using worst-case traffic simulation to limit the bus-load under 50% of the maximal bandwidth to guarantee the quality of service and relatively predictable end-to-end delay time. In some cases, the so-called “private CAN” – a dedicated short-range CAN is used to connect ECU with intelligent sensor and actuator.
5. Redundant CAN-communication with duplicated physical transfer paths or redundant messages on the same CAN. Here a typical trade-off between the performance and dependency should be made, since the redundant messages should be synchronized and compared on the receiver on cost of the real-time.
6. Use reliable control loop and fault detection filter to make the system more reliable against communication fault.
7. Intelligent algorithms to tolerate the timing drift of CAN messages.

While the methods 1 to 5 are relative self-explained, the methods 6 will be discussed in (process safety as discussed in the example of Subsection 9.3.1.3) as a software-based dependability service, the method 7 is demonstrated here with a detailed example.

To solve the problem in Figure 8-4, some intelligent mechanisms as following can be applied. One possible way is to synchronize the send/receive execution by trigger signal from receiver (time-synchronization with event-triggered bus). Another possible approach is to check integral of variation of sensor values $\Delta(S_1 - S_2)$ over a defined time period. With the proper choose of monitoring period T, the integral of sensor signal offset $\int_T \Delta(S_1 - S_2)$ (comparable with the integral of white noise in the telecommunication within a sufficient long period of time) should be under a limited threshold in faultless situation.

As introduced in Subsection 3.3.4.2, FlexRay, as a time-triggered protocol, is the in-vehicle communication network with the highest dependability potential. In the coming two or three years, more and more high-end cars from European OEMs will be running on road with FlexRay (the new

BMW X5 has been equipped with FlexRay for adaptive drive application in chassis since December 2006). The trend from CAN to FlexRay is triggered by the two of most superior aspects: bandwidth and dependability. FlexRay will find its way first in the safety relevant chassis applications or as backbone to provide required large bandwidth in chassis domain. The substitute of CAN-interfaces with FlexRay and to maintain a hybrid communication system of CAN and FlexRay will require additional effort, but it can be foreseen in the near future this effort will be compensated by the economy of scales.

As mentioned in the design rule 4 in Subsection 8.1.1, the safety feature of communication bus provides only the physical data integrity between the communication controllers, however, it provides no guarantee between the application-to-application data integrity service. The topic about application-CRC to ensure the application to application communication will be discussed in Subsection 9.3.1.3.

8.3 Concepts of ECU hardware architectures

Emphasis in this subsection is firstly the concrete design of dependable ECU architecture. In the second part of this chapter, hardware architecture of microcontroller components to improve the dependability will be discussed.

8.3.1 Dependable architecture driven by the safety integrity requirements

The decision for the dependable HW-architecture is mainly driven by the requirement of safety integrity level of the system. While the fail-safe and fail-operational units (1oo1D-, 2oo2-, 2oo3- and dual-duplex-system discussed in Subsection 3.3.3) are categorized according to the fault-treatment strategy, in the praxis, the decision of hardware architecture is mainly determined by the compliance with the required safety integrity level and the fulfillment of the system safety criterion.

That is to say, the desired safety level should be reached with the help of the dependable hardware architecture and software dependability services. A quantitative safety case (e.g. SFF and PFH) should be given to the applied dependable hardware architecture to support the decision of hardware architecture (with e.g. FMEDA or quantitative FTA).

According to IEC61058 (see Table 3-2), SFF with at least 99% is required by ASIL-C and ASIL-D safety application. Experience from praxis has shown that, if such a system should be built up with one-channel system (e.g. 1oo1D system), considerable effort should be carried out to tolerate hardware transient and permanent faults (e.g. CPU computation fault, data fault in RAM as discussed in the hardware fault types in microprocessor core and supervisor of Subsection 6.2). The monitoring of such hardware faults should be implemented with software monitoring (e.g. software diversity of different algorithms). While the software diversity can not guarantee a 100% fault detection as well, because some low-level hardware faults can influence the both independent control algorithms (common cause). As a clear trend [μ CI06], in the safety relevant automotive applications, architecture with dual-core (the 2nd variant of Dual-Duplex-Processor ECU as shown in Figure 3-7) has aroused the most future interest. In the following, further research of 2oo2D-system and 1oo2D-system will be carried out.

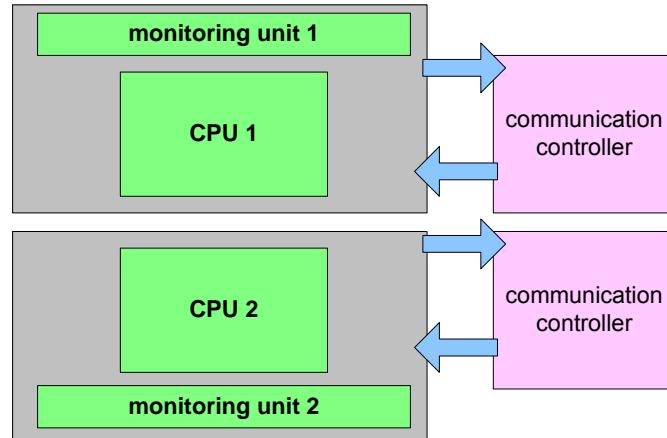


Figure 8-5: Variant of Dual-Duplex-ECU as 2oo2D-system

It can be seen that the 2oo2D.system in the simplest case as shown in Figure 8-5, is a replication of 1oo1D-structure without hardware diversity. Because the control of actuator over the communication system is redundant and no influence from one channel to the other is allowed, such a system can only guarantee an improved availability but not a high quality of system diagnosis coverage. SFF with at least 99% is difficult to reach with real components and design shown in Figure 8-5, because an unknown fault in any channel (e.g. HW-failure) can lead to a safety relevant failure of the whole system.

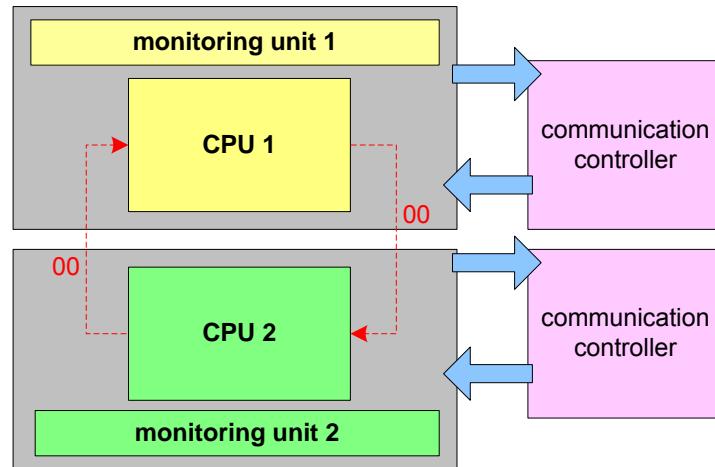


Figure 8-6: 2oo2D-system with diversity in hardware

In order to reach the ASIL-D requirements with 2oo2D hardware architecture, the two 1oo1Ds should be implemented with hardware diversities. According to [IEC02] Table A16, A17 and A19, hardware diversity is recommended as highly effective to control the systematic fault (e.g. hardware computation fault). Each subsystem should have DC of 90% as required by ASIL-B. Mutual monitoring between the two CPUs should be implemented as well.

As depicted in Figure 8-6, in this architecture for example, system 1 can be an ASIC microcontroller while system 2 can apply FPGA technology. Here design diversity can be also

used, e.g. system 1 and system 2 are both ASIC but from different semiconductor producers. Because of the hardware diversity, low level hardware near software should be implemented in a diversity manner as well. The mutual monitoring and comparison of computation results are depicted with red dashed lines, which can be realized with direct µC I/O or external communication bus.

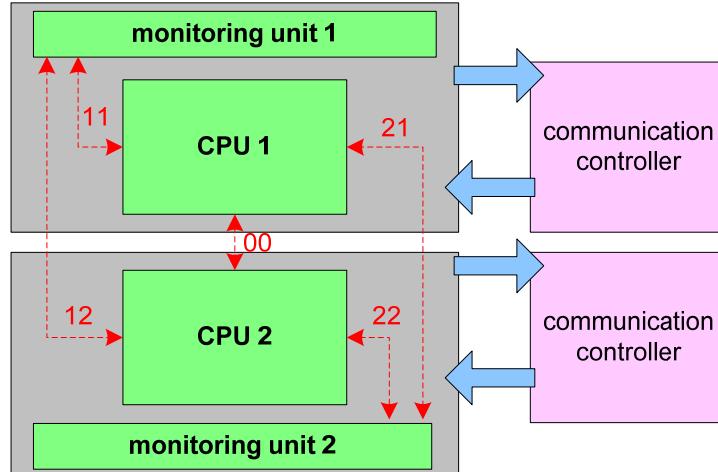


Figure 8-7: Variant of Dual-Duplex-ECU as 1oo2D-system

According to the safety norm IEC 61508-6 Annex B [IEC06] and IEC 61508-2 Subsection 7.4.3.1.6 [IEC02], the 1oo2D-system shown in Figure 8-7 with self-test is highly recommended for ASIL-C and ASIL-D systems, which requires a high SFF of 99%. Compared with 2oo2D-system, 1oo2D-system lets the monitoring units and CPUs communicate with each (as depicted with the red signal paths). The dual-core CPUs compare the intermediate results and final results with mutual agreement (signal path 00). Each CPU is not only monitored with local monitoring unit but can be monitored as an option remotely (e.g. signal paths from monitoring unit 11 to CPU1 and 12 to CPU2). If a remote fault is detected, such an additional signal path can be used to passivate/deactivate the neighbored processor unit in order to keep the whole systems running in a fail-degraded mode.

The signal paths here (00, 11, 12, 21 and 22) can be implemented with direct microcontroller I/O interfaces or with the external bus-communication or as a hybrid communication system of both. When the both CPUs are realized on the same silicon, the communication here can be implemented with shared memory or internal communication bus. In case of inconsistency the neighbored channel can be shut down. Thus the 1oo2D-system guarantees both high safety and availability.

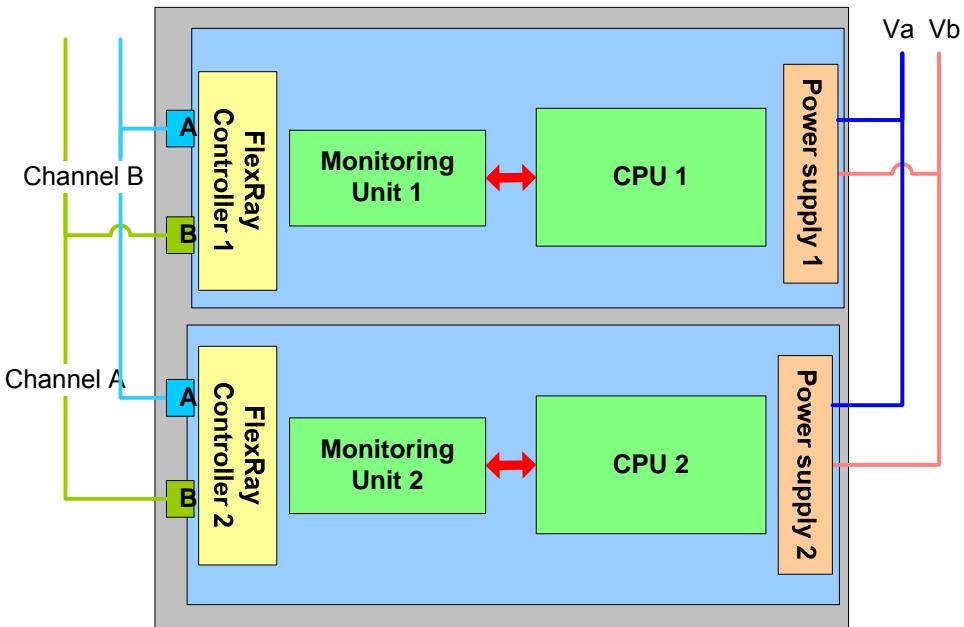


Figure 8-8: Implementation of Dual-Duplex-ECU as 1oo2D-system [HWAA06]

An implementation of Dual-Duplex-ECU as 1oo2D-system in praxis is shown in Figure 8-8, where the fail-operation unit is made up of two fail-silent units. The communication within the fail-silent unit (between the monitoring unit and main processor) is implemented on board. While the communication between the two main processors and the possible remote monitoring of main processor is implemented with dual-bus FlexRay communication. It is to mention here, the dual-bus communication is only an option, which depends on the overall in-vehicle network architecture. The single bus solution with end-to-end communication services in Subsection 9.3.1.3 can applied as well, as long as the PFH of the communication system as 1×10^{-10} as discussed in 7.2.2 ISS EP 2.4 is fulfilled.

It is to mention that the 1oo2D-system can be implemented with two units on one silicon (dual-core) or dual-processor (two separated CPU). Generally speaking the first approach of dual-core may share more resources, e.g. integrity checking information such as counters, CRCs, program flow information and memory management, etc. The intra- μ C communication can be implemented more easily.

As a summary, in order to reach the aimed the ASIL-D requirement, the following dependable hardware architectures can be applied:

1. A 2oo2D systems with two 1oo1D (one channel) system as depicted in Figure 8-6.
2. A 1oo2D system with a dual-core system without diversities in software (as depicted in Figure 8-7 with aimed SFF of 99%)
3. A 1oo1D system with diversity in computation units (e.g. integer and floating computation unit) and intelligent software monitoring mechanisms of aimed SFF of 99%.

Innovative safety applications with ASIL-D requirement can be developed by the premium OEMs with the 1st approach (redundant hardware of diversity concept). Such applications are seldom

provided as serial product and are usually only available as an optional feature with a limited production volume. The short development cycle of the safety innovations with this hardware architecture brings the competitive advantages needed by the premium OEMs. An example here is the hardware architecture of the first generation of Active Front Steering System [Rei05], in which the safety relevant functions (main function and monitoring unit) are computed on diverse processors (MPU and NEC microcontroller) with mutual monitoring.

With the experience gained from such a pilot project and the requirement of continuous optimization of the system, when this safety application is provided as serial equipment or even considered by the middle-class OEMs, the safety architecture of the 2nd approach can be considered. The systematic hardware and software fault should be managed by safety process in the 2nd approach. The coincident common cause faults resulted from hardware or environment can be managed by the dual-core hardware architecture. With dual-core architecture it is also easier to analyze and proof functional safety. As long as the quantitative safety requirements to the SFF and DC can be reached in the safety case, no diversity in software development is required. The additional cost in hardware here can be compensated with the economy of scale production, simplified assembling and maintenance, thus the second approach can be preferred for the safety relevant systems by the premium OEMs for the volume production.

The 3rd approach is the simplest solution from the view of hardware architecture. From the hardware side, HW-Watchdog, as discussed in Subsection 8.3.4 with enhanced features such as intelligent algorithms to watch the program control flow and check the plausibility of computation results should be applied. On side of software, significant effort is required to achieve system dependability, e.g. each safety relevant signal should be monitored with diverse software mechanisms as to the value and gradient, safety relevant functions should be computed on the diverse computation unit (e.g. integer and floating) in one microcontroller. The ASIL-D safety functions should be implemented with software diversity, e.g. C and model-based approach with two independent development teams. The significant development effort in software is fortunately not proportional to the production volume. Thus the third approach can be preferred for the safety relevant systems by the volume OEMs.

8.3.2 Monitoring of sensor and actuator components

As discussed in Chapter 7 IEP Step 2.4 about mapping of system-PFH to FAA-components, about 75% to 90% system failures lie in the input (I/O, sensor and communication system with outside) and output (control of actuator). The monitoring of ECU input and output provides a significant potential to reduce the PFH of the whole system. However, the monitoring of sensors and actuators is quite application specific/dependent and there are less platform independent strategies. In the following, only a few examples from praxis are discussed.

The easiest approach is to use the redundant sensors to check the plausibility of the signals as shown in Figure 3-3. Asymmetric redundancies can be also applied, in which low-priced sensor (eventually with another principle of physical measurement) is used as a slave to monitor the main sensor. The input of the sensor should be checked in the time and value domain (with gradient/threshold check, message counter, CRC, etc). On the consumer side of the sensor signals, ECU can compare the received signals with other input values to check the physical plausibility of the sensor signals, e.g. the plausibility check of the gas pressure to its temperature.

By the monitoring of actuators we concentrate on the mechanisms of fault detection and deactivation of actuator for the fail-silent strategies. An example is depicted in Figure 8-9, in which microcontroller controls the electronic motor via power electronics. By monitoring different physical parameters of power electronics as well as motor (e.g. position and rotation speed of the rotor, voltages/current at power electronics and intern voltages/current at motor coil), microcontroller can disable the actuator via different independent “enabling” links to power electronics. The enabling-paths of actuator (to power electronics and directly to motors) should be implemented redundantly. Only at a positive output of the AND-logic of all enabling-paths, the motor can be activated. The different enabling paths should be checked at system initiation phase, the last operational mode should be saved during shut-down phase as well. By measuring the key-parameters of motors (rpm, coil current and voltage, rotor position, etc.), the microcontroller can calculate the expected parameter of the actuator for the next monitoring cycle with a simulated motor-model. By comparison of the current physical value with the results from the simulation model, the running behavior of the motor can be monitored.

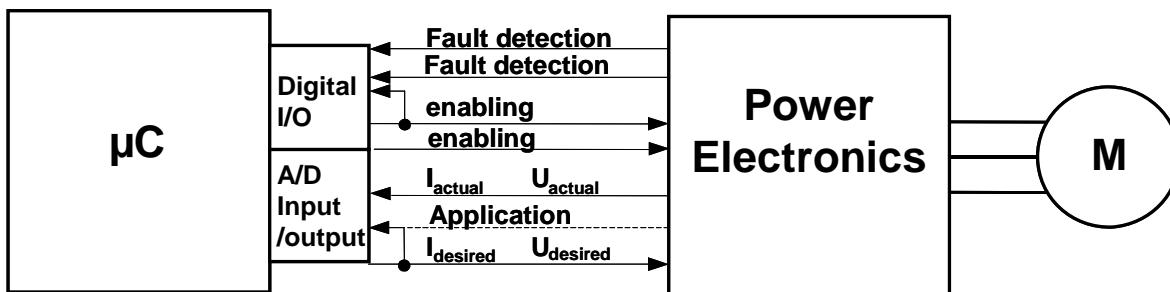


Figure 8-9: Example of the actuator monitoring (electronic motor)

8.3.3 Memory protection

As a state of the art method to protect memory area from EMV or other coincident influences from environment, there are following mechanisms to check the consistency of the memory area.

Invariable memory ranges

- CRC Signature: one word (8-bit CRC) and double word (16-bit CRC) signature are recommended in IEC 61508 – 7 A3.4 [IEC07]. The effectiveness of the signature depends on the width of the signature in relation to the block length of the information (pay-load) to be protected (at least with a hamming-distance of 2 [DIN93]).
- Block replication: In order to prevent common cause failure, safety relevant data should be saved redundantly in different areas of memory.
- In different phase of system states (e.g. start-up, operational mode and shut-down), the phase specific data along with CRC should be saved (e.g. at the end of each phase or a periodic test during the operation).
- In PROM, which contains executable code, all unused area of memory should be written with a fail-safe data value, e.g. an illegal operation code, which triggers a fail-safe interrupt.
- For the aim of diagnostics and filter of non-systematic fault, all the fault detection in the memory range should be recorded in a fault-detection counter.

- For each “write” operation to the memory, the written data should be read and compared directly after the write operation.

variable memory ranges

- With Parity-bit for RAM only a limited DC can be reached.
- RAM monitoring with a modified Hamming code or detection of data failures with error-detection-correction codes (EDC)
- Double RAM with hardware or software comparison and read/write test.

With the trend of increasing density of applications, how to prevent the undesired mutual influence of application components is an important topic for the ISS-application. While the design and software service to improve the system dependability will be discussed in Subsection 9.3.2, from the hardware side, it should be guaranteed that no non-authorized or faulty application components should allocate system resource assigned to the other applications. By means of system resource the following objects are considered: memory space, CPU time and permission to allocate system object like error-hook and interrupts. The protection of memory space brings new requirement to the ECU hardware architecture to have an additional Memory Management Unit (MMU) and Memory Protection Unit (MPU).

One of the preconditions for the memory protection is the memory portioning, so that each parallel running application can be assigned to an individual memory area. In this way, each task can be strictly controlled in the RAM memory that can be accessed. To avoid the waste of memory resource, the allocated units should be divided small enough to fit the requirement of a normal task of the automotive embedded system. The tasks themselves should be categorized as trusted task or non-trusted task. Each of them has their own private memory areas, while non-trusted task can not write or use call-back function to access the private area of other tasks. The memory protection unit controls the position and range of addressable segments in memory and the read/write operations allowed within addressable memory segments. Any illegal memory access can be detected by the memory protection hardware, which then invokes the appropriate service routine to handle the error.

By means of this, non-trusted tasks are the safe object, which are controlled under the OS all the time, while the number of privileged trusted-objects should be limit as small as possible.

8.3.4 Hardware watchdog monitoring

The hardware watchdog ([GAN03] and [LAN97]) is the last line of defense when the code collapses, in which hardware watchdog timer is the state of the art. The hardware watchdog timer should be prompted periodically by the monitored object (writing a “service pulse”) to prevent a hardware reset. The intention is to bring the system recovered from the hung state into normal operation. Usually the hardware watchdog is a co-processor or external parallel running ASIC-chip to the main processor.

Traditional hardware watchdog has a separated time base as the main CPU to monitor the activities in an adaptive time window, such hardware watchdog has only a low diagnostic coverage. By the so-called question/answer mechanism, the hardware watchdog can trigger/challenge the main CPU to check if it works logically correctly. In this case the logical sequence of program flow can be checked as well. By combination of these both mechanisms, the temporal and

logical monitoring of program sequences, a high diagnostic coverage can be reached. Thus such an intelligent hardware watchdog is recommended by IEC61508-2 [IEC02] to improve the system diagnostic coverage.

8.4 Design use-cases of ISS hardware architectures

In the following subsections, the distribution and mapping of ISS-applications to the system topology is demonstrated with a few examples, following the design guideline proposed as above.

8.4.1 Distribution of ISS applications to vehicle domains

A near-future scenario of ISS-applications as specified in Subsection 1.1 is illustrated in Figure 8-10. This figure also describes the hierarchy of the different ISS-functions, starting from the elementary physical sensors and actuation devices (bottom level) up to the high level integrated application (top level, what the customer can experience as safety application).

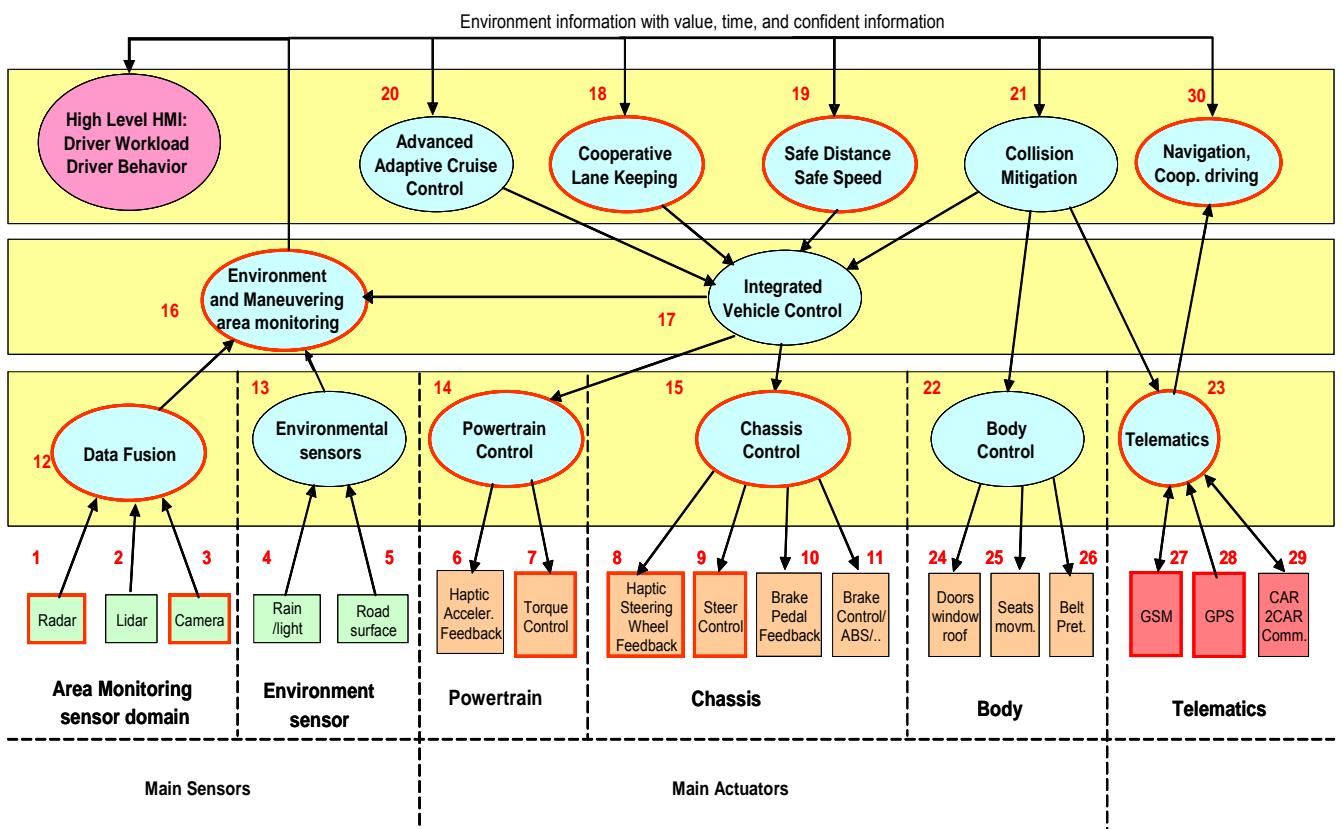


Figure 8-10: Examples of ISS application and mapping to the vehicle domains

Here we focus on two use-cases of ISS applications, the SafeLane and SafeSpeed. As highlighted with red color in Figure 8-10, SafeLane is a lane keeping assistant application while SafeSpeed is an ISS-application to automatically limit the vehicle speed to an externally commanded maximum value. The present example ISS application can be mapped into the following four different vehicle domains:

- Chassis Domain

The SafeLane function mainly uses camera information and steering wheel sensors as environment information source. Additionally, it may use information coming from the Adaptive Cruise Control for monitoring the road in front of the vehicle and from the chassis units for understanding the vehicle dynamics. The function can provide haptic feedback through direct actuation on the steering wheel.

Steer-by-wire can be applied here to cooperate with SafeLane, it performs the vehicle steering function by means of reading the drivers demand (steering wheel sensors) and by driving the appropriate actuators (wheel steering motor) of the steering system.

- Telematics Domain

SafeLane and SafeSpeed consider infrastructure information elements, such as road traffic, digital road maps, precise vehicle positioning and local speed regulations.

- Powertrain Domain

SafeSpeed may interact with the powertrain and chassis domain by requesting a reduced speed (reduce the engine torque and brake) or by providing a haptic warning to the driver through the accelerator pedal.

- Body Domain

All the functions must interact with the driver via displays and acoustics for warnings and information about operational modes. All the functions should provide an integrated diagnostic system service.

In Figure 8-11, a possible mapping of ISS-Application SafeSpeed to the chassis and powertrain is demonstrated. SafeSpeed uses both engine- and brake-control to adapt the vehicle speed to the environment (e.g. telematics). Actually the distribution of ISS-application to vehicle domains is the mapping to of FAA to the hardware architecture, which generates the first FDA-model. Following the mapping rule in Subsection 8.1.2, the brake ECU and engine ECU are reused. At the first glance, the introduction of SafeSpeed brings a new ECU (as the prototype ECU with MABX in Figure 8-11), but for the serial development the SafeSpeed function can be integrated on an ECU for Integrated Vehicle Dynamic Control or Brake Management ECU, which has the similar safety integrity.

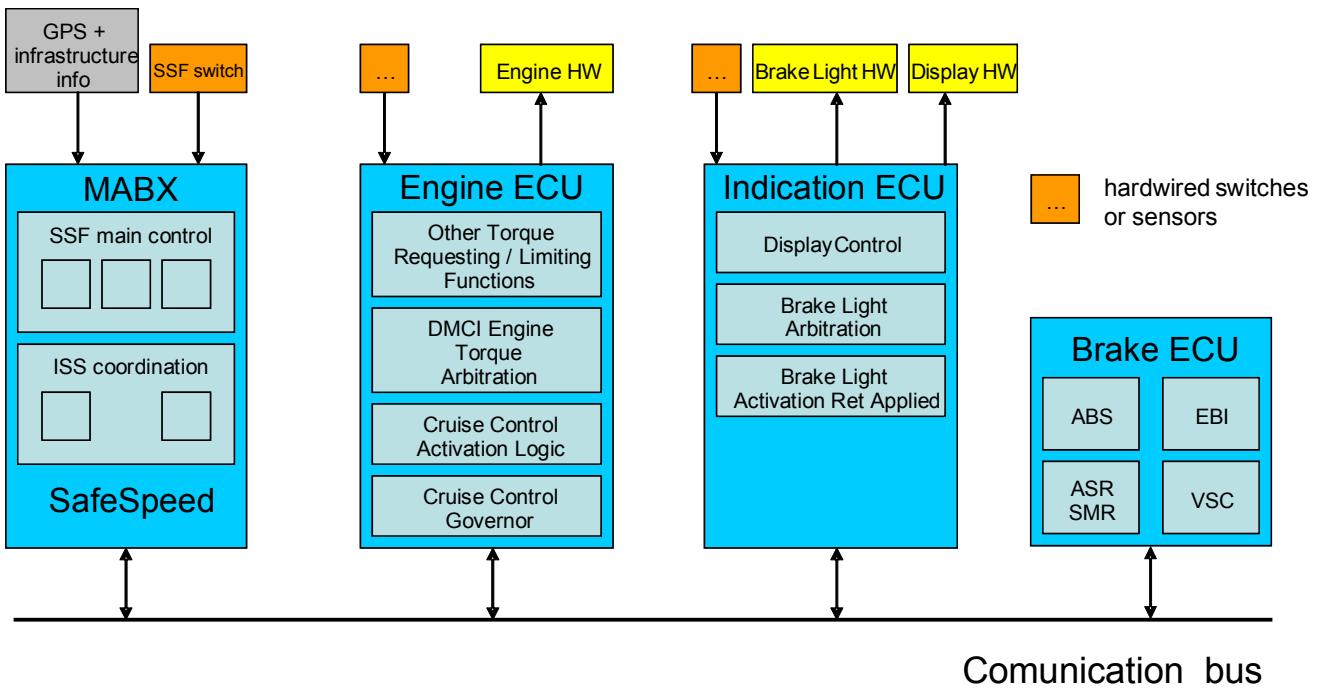


Figure 8-11: Distribution of SafeSpeed to the HW-architecture (FAA → FDA)

8.4.2 ISS system topologies and communication systems

After the discussions above about concepts of communication system, ECU hardware architectures and concepts about system topologies, esp. system design rules in Subsection 8.1.2, a few concrete examples about vehicle system topologies of ISS are given as following:

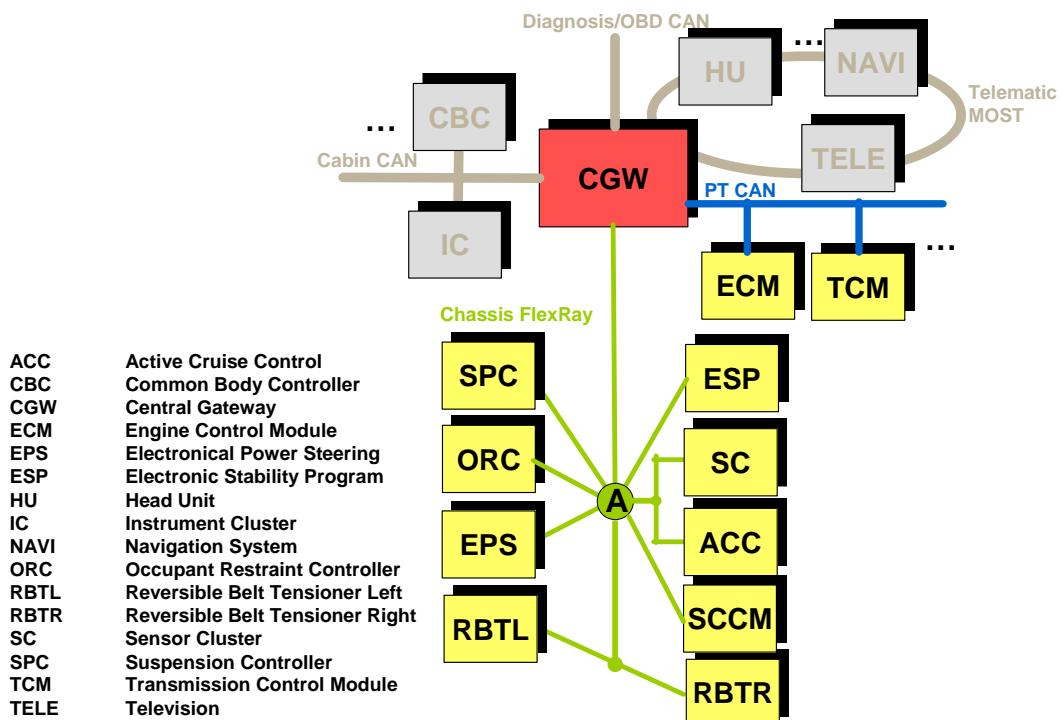


Figure 8-12: Exemplary system topology of future mid size car [HWAA06]

In-vehicle system topology shown in Figure 8-12 is a practical implementation of the central gateway concept for the mid size cars shown in Figure 8-2. The integrated safety applications are distributed among the ECUs, highlighted in yellow and connected with chassis FlexRay and powertrain CAN. A single channel FlexRay is applied here to enable the time triggered communication between the chassis nodes with a star topology with help of FlexRay active star [Ele04]. The system topology here is mainly designed for ISS, which require the fail-silent/fail-safe behavior with the mechanic back.

A good example here is a dummy “SafeSteering” ISS application. Compared with “SafeLane” as a lane keeping assistant system, SafeSteering can increase vehicle stability by interference the steering and vehicle dynamic actively. A practical example here is the superposition of an additional actively controlled steering angle to the driver input as introduced in Audi Active Dynamic Steering [ADS07]). In low speed area, it can provide the driver with park assistance support as well. SafeSteering is an ASIL-D ISS-application without any comparable system from the praxis, which is used here to demonstrate the dependable hardware architecture.

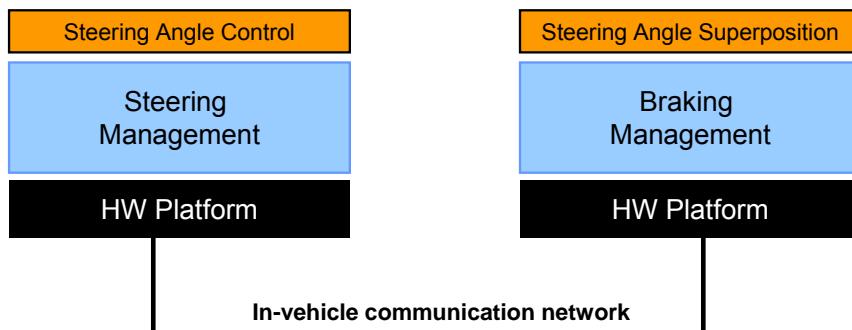


Figure 8-13: Communication and functional architecture of SafeSteering

As depicted in Figure 8-13, the SafeSteering is distributed between two main function units, the Steering Management Unit (SMU, as an electronic steering system of semi steer-by-wire technology with mechanical backup) and Braking Management Unit (BMU, as an enhanced “ESP”). The SMU provides the BMU with current wheel steering position (based on the information from actuator, e.g. rack position and rotor position of electronic motor), driver command in steering wheel angle and steering torque. BMU can provide SMU with the suggested additional steering recommendation with steering torque support or superposition angle according to the different driving situations, e.g. parking and highway.

Theoretically the computation of superposition angle and assistant steering support can be mapped to both SMU and BMU, since they are of the similar safety integrity of ASIL-D. The communication between SMU and BMU can be implemented with CAN or FlexRay as long as the ASIL-D requirement of communication is reached.

Compare the Figure 8-13 with Figure 8-12, the SMU is comparable with EPS, while the BMU, in the simplest case, is an ESP. Following the design rule 1, since most of the vehicle dynamic data needed are available on SMU, the computation of supposition angle can be mapped to ESP as shown in Figure 8-12, while the result of superposition angle can be sent to SMU. By detection of any safety relevant fault in SMU itself, the assistant steering moment /superposition angle of SMU should be switched off completely with a reduced ramp. A mechanic steering system is available in

the fail-degraded situation, so that the driver can still control the vehicle. At the same time, the communication from SMU to other systems (e.g. sending-out steering-wheel-angle to ESP) can be passivated in fail-silent mode.

If the communication channel between BMU and SMU is implemented with CAN, a weak chain can be identified in the end-to-end communication for the steering recommendation. In order to reduce the safety requirement for CAN, as discussed in the rule 10, the superpositioned angle/additional steering torque should be limited in range along with the process-safety discussed in Subsection (Subsection 9.3.1.3), so that even when the signal is faulty, the vehicle steering can be still controlled by the driver.

In case of a complete communication break down, SMU should be able to work in a stand alone mode. For this reason, the steering wheel sensor and actuator are connected directly to the EPS in Figure 8-12.

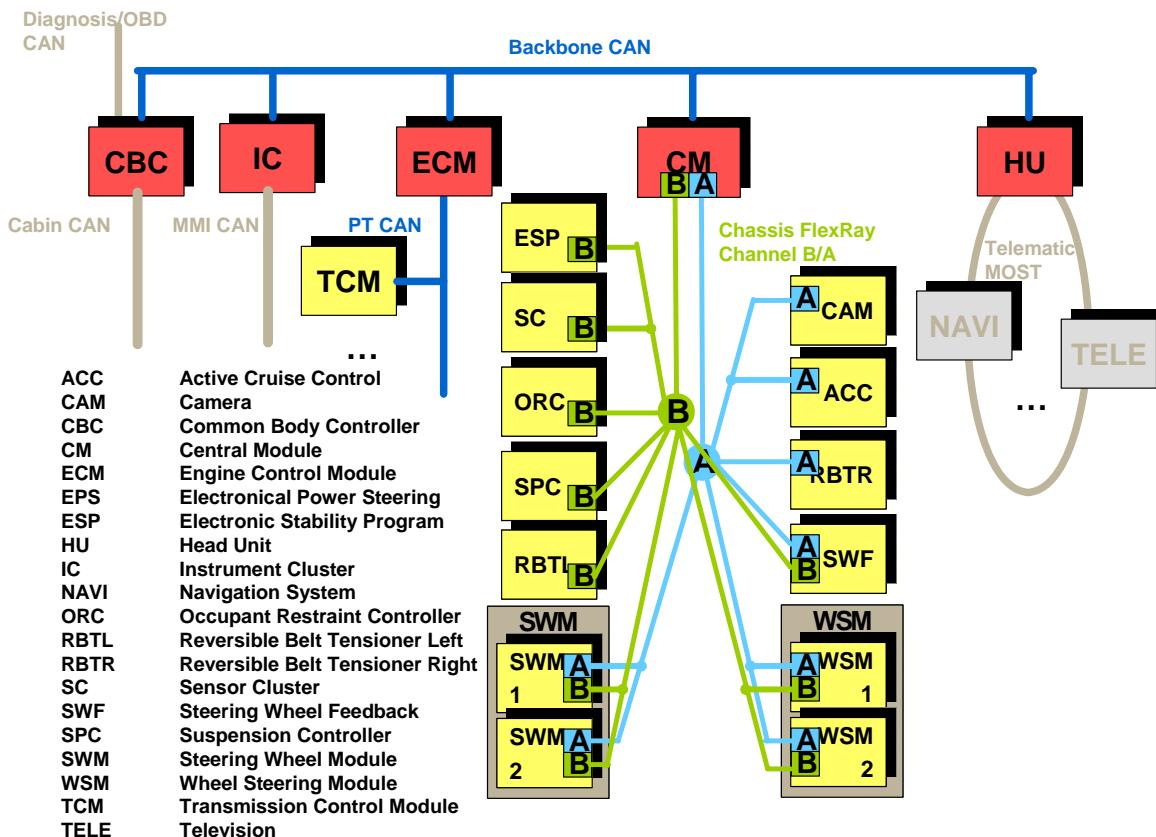


Figure 8-14: Exemplary system topology of future large size car [HWAA06]

Figure 8-14 shows an example of system topology of future large size luxury vehicles with steer-by-wire technology. Without the mechanic backup, the steering function here requires a fail-operational behavior.

Take again the dummy “SafeSteering” as an example: now this ISS is distributed on four nodes, SWM (Steering Wheel Module), ESP, CM (Central Module) and WSM (Wheel Steering Module), which are connected with a dual-channel FlexRay. One possible mapping is to map the steering management function to WSM, while the braking management function is mapped to ESP. The advanced steering/braking assistance and recommendation can be mapped to CM.

Here, the nodes with the highest safety integrity are the CM, SWM (sensor) and WSM (actuator ECU), because they are directly involved with Steer-by-Wire. These nodes are connected to both FlexRay channel A and channel B (partly redundant FlexRay with active stars). While the other less safety relevant nodes such as ESP, SC, ACC, CAM, etc. are only connected to single channel FlexRay A or B with their relevant sensors and actuators.

In the CM, different integrated safety functions and other chassis functions can be merged (sensor fusion) on one ECU to have an integrated vehicle dynamic control. The CM is responsible to give the SMU and BMU (ESP) the additional steering recommendation and braking recommendation.

It is to note that here only the SWM and WSM are redundant in 1oo2D architecture and connected with both FlexRay channels, while the CM is not redundant. In case of any problem with CM, the basic electronic steering function is still available with SWM and WSM. The SWF (Steering Wheel Feedback) is not implemented in a redundant architecture either, because the feedback force can be still provided with mechanical spring force.

CBC (Common Body Control), IC (Instrument Cluster), ECM (Engine Control Module), CM and HU (Head Unit) in Figure 8-14 are connected with a backbone CAN.

8.4.3 ECU hardware architecture for ISS

The hardware structure of SMU discussed in Figure 8-12 can be designed as a 1oo2D dual-core system. Figure 8-15 depicts one of the possible hardware architectures (adapted from the approach 2 as discussed in Subsection 8.3.1). The 1oo2D architecture is integrated on a dual-core chip. The same data input will be provided to the both dual-core processor with an intended short time delay, so that the comparing unit can compare the computation results from both CPUs step-by-step. In case of computation fault, one of the enable-path of the actuator control will be disabled.

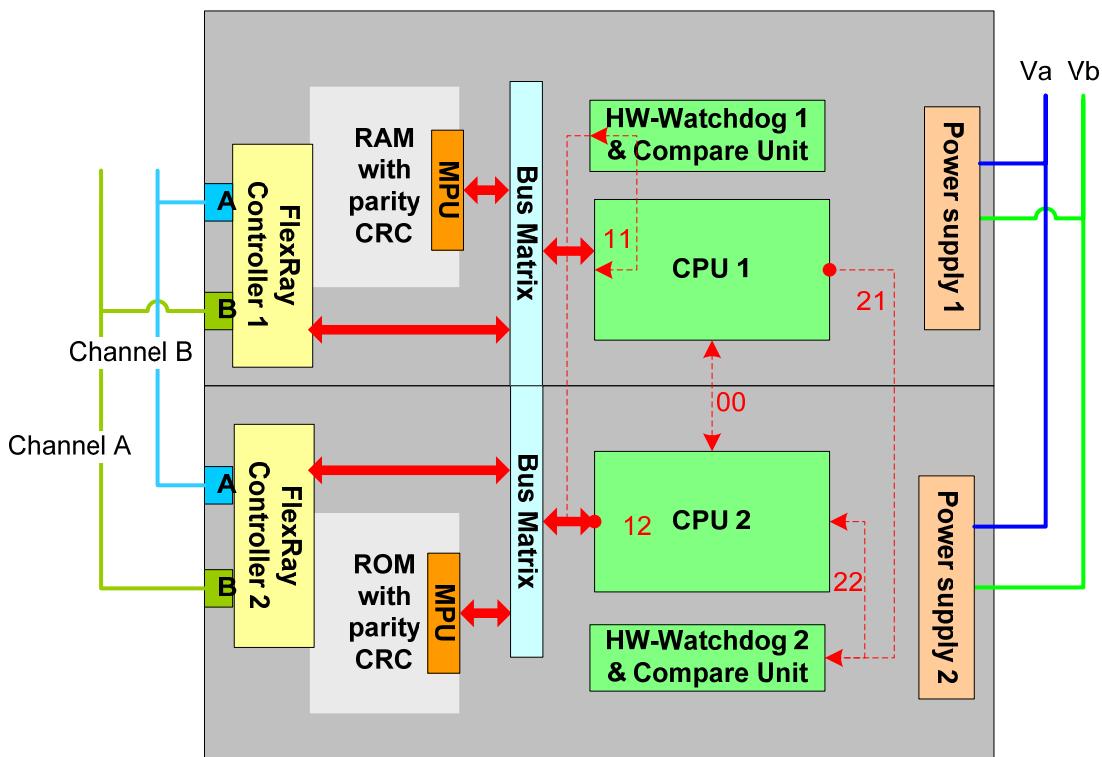


Figure 8-15: Exemplary 1oo2D dual-core ECU architecture

Since the both cores are built on one silicon board, the access to the memory is controlled by the MPU with the help of a bus matrix for the high-speed internal data exchange. In order to tolerate common-mode faults, both processors have independent memory and power-supply. RAM and ROM are cyclically checked with parity and CRC, etc. The redundant FlexRay controller to both FlexRay channel A and channel B is only an option. Single channel FlexRay with one transceiver can be applied as well as long as the safety requirement of system can be reached.

8.5 Conclusion of hardware architecture framework

Guided by the safety integrity requirements as a red line, the design of ISS hardware architecture reflects again the distributed development between OEMs and suppliers.

OEM defines the overall system safety integrity requirement, topology and communication networks for ISS, while taking the distribution ISS-application into consideration. A balanced design considering different constraints and requirements (safety as the most important among all) will make the system integration more effective and improve overall system dependability significantly. The suggested system topologies and guidelines provide a framework for this decision phase.

On the other side, suppliers should provide the safety concepts in the hardware architecture, which fulfill the safety requirements from OEM. With the platform strategy and standardization, system suppliers are more and more involved in the early design phase to contribute their know-how by the definition of ECU architecture. Since the quantitative safety requirement is always a system wide term, the optimization of hardware architecture (e.g. evolution of safety concept from hardware diversity to dual-core) should be accompanied with intelligent software based dependability services.

9 Software platform for the Integrated Safety Systems

9.1 Trends of software platform – a benchmark with IT-industry

As mentioned in the state of the art about trends and challenges in the embedded in-vehicle software architecture, there is a clear trend towards standardization of a common middleware. From this point of view it is quite interesting to take a look in the history of IT-industry about the development of operating system. Similar like the OS in embedded world, OS in IT provides a set of functions needed and used by most applications and the necessary linkages to control a computer's hardware [HOS07].

The early computers lack any form of operating system and first original operating systems on computers go back to the early 1960s. Early operating systems were very diverse, with each OEM producing one or more operating systems specific to their particular hardware. Every operating system, even from the same OEM, could have radically different models of commands, operating procedures, and such facilities as debugging environment (quite similar with automotive embedded electronics now). Typically, each time the manufacturer brought out a new machine, there would be a new operating system. This state of affairs continued until the 1960s when IBM developed the System/360 series of machines which all used the same instruction architecture. With the rise of minicomputers, the UNIX operating system was developed at AT&T Bell Laboratories in the 1970s, which was conceptually the same across various hardware platforms. It was with the introduction of 8-bit home computers and game consoles of 1980s that real operating system for private customer was introduced. The decreasing cost of peripheries and processors made it practical to provide user more complicated operating systems with enhanced features.

When we compare this history with the development of embedded in-vehicle software, some similarities could be identified. The state of the art in the software platform in embedded in-vehicle software is somehow unfortunately comparable to the situation of IT-industry in 1970's. While a common operating system (OSEK, etc.) already exists, it is still not applied in each vehicle domains by each OEM. Each automotive OEM has his own adapted operating system, its own standard software modules. Automotive tier-one-suppliers esp. system suppliers are providing their ECUs along with their specific house-made software and hardware. The migration process of the systems for another customer (OEMs) requires a lot of effort. The tier-two/three suppliers (e.g. semiconductor and ECU hardware providers) are still developing the hardware along with their own drivers and low basic software modules, which again have a poor reusability for the tier-one suppliers and OEMs.

But why do we have this ca. twenty years' distance to the development of IT-industry? On one side the introduction of automotive electronics began only at the early of 1980's (as depicted in Figure 1-1), on the other side, because automotive industry has been traditionally strongly shaped by the mechanical engineering, which has quite different requirements and constraints compared with IT-industry. Thus the embedded in-vehicle electronics can be only a slower-follower of IT-industry.

Based on this benchmark, a conclusion can be drawn that the same development as in the IT-industry for the future development of embedded in-vehicle software will happen in the near future. This evolution could be slower than IT despite of lesson learned from IT, however, with a quite clear and steady similar direction as in IT.

Taking the Moore's Law and cost-awareness of automotive industry into consideration, the hardware devices and peripheries are quickly becoming cheaper, while more and more functions will be implemented with software. Since ECU software platforms (including: hardware-drivers, operating system and standard software modules) can not be experienced by the customers directly, they are not really competition relevant for the OEMs, and neither for the Tier-1 suppliers. Standardized open software architecture will help to fix software bugs, improve the maturity of basic software modules and gain the experiences from integration of application software components with the underlying software platform. For the safety relevant applications, as have mentioned before, the same trend to have a software platform with dependability software services will take place. The standard library of dependability software services can be commonly applied by different OEMs and suppliers. This will surely contribute a lot to improve dependability of the whole system.

9.2 Concepts of dependability software architecture

As introduced in Chapter 5, in the EU-funded industry partnership EASIS, the first for the Integrated Safety System designed software architecture was developed. The software architecture framework for ISS is illustrated as Figure 9-1.

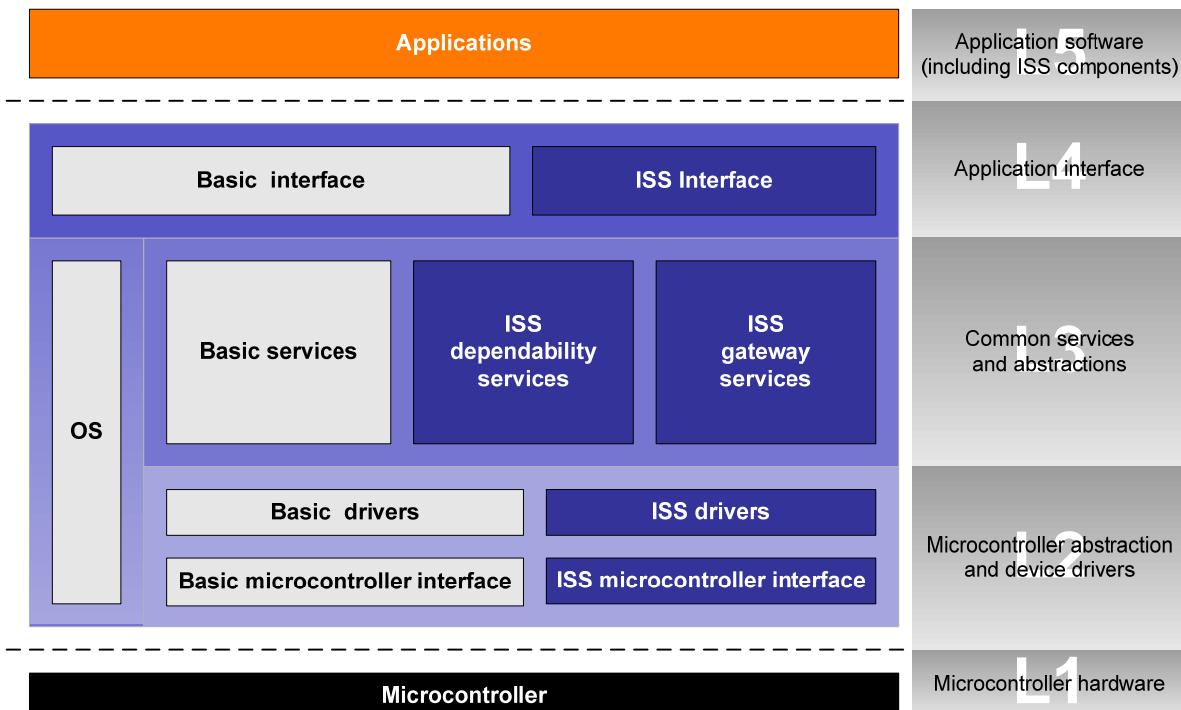


Figure 9-1: EASIS layered software topology [SWP06]

In Figure 9-2, the EASIS software topology with an overview of embedded basic software modules and dependability software services [HIL06] is given, where the software services with * on the symbol of package are optional and configurable to the safety requirements of the applications.

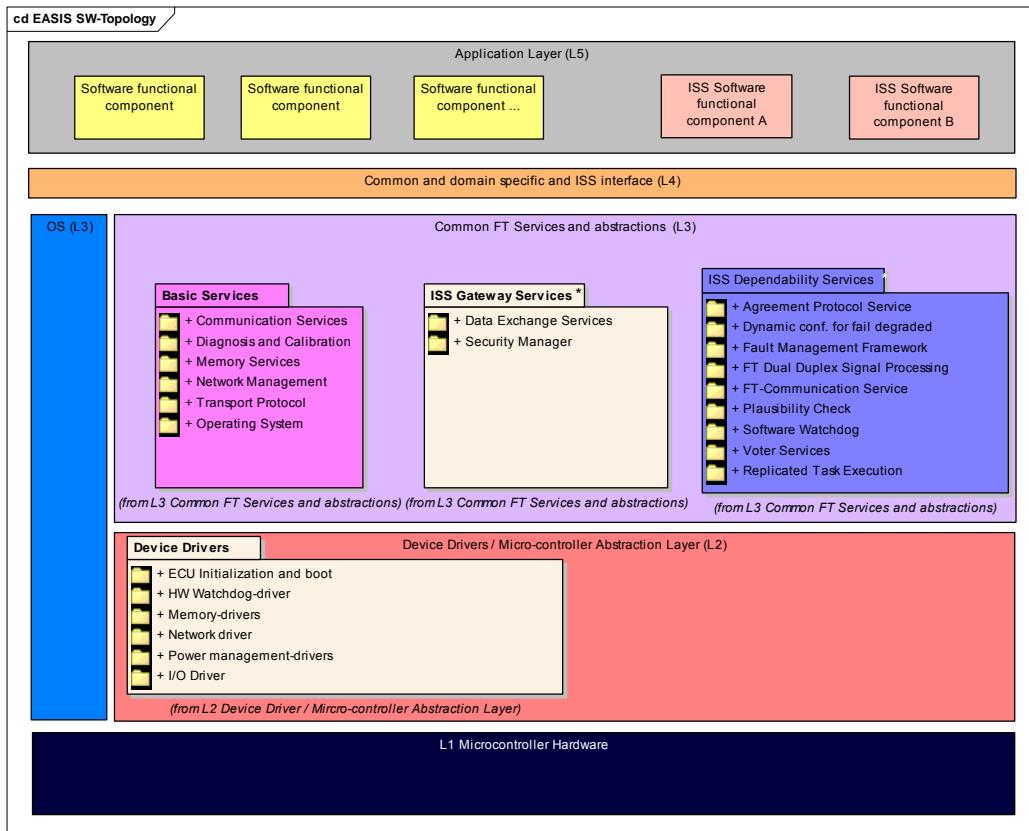


Figure 9-2: EASIS software topology –software components and services [SWT06]

In the following the layered software architecture framework depicted as Figure 9-1 is specified in details.

Layer 5 (Application Software): On the top of the EASIS layered architecture is the Application Layer. This layer harbors all application software components that provide the “actual” functionality that the system/ECU is to provide (i.e. the payload functionality). These functional components use the common/domain interface to access data and other items contained in the platform. The functional components can be domain-specific components, e.g., providing algorithms for powertrain control, or they can be cross-domain ISS components. In the application layer, software functional components with different dependability requirements can be integrated onto one ECU, including non-safety relevant functional components.

Layer 4 (Application Interface): Between the Application Layer L5 and Service Abstraction Layer L3 is the Common/Domain specific and ISS Interface Layer. It serves as application interface, which contains an interface for the functional components of the upper layers and is therefore the platform upon which all functional components are built. This is the ultimate abstraction layer of all underlying layers. It is worthy to mention that layer L4 keeps conformance to the AUTOSAR Runtime Environment (RTE) layer.

Layer 3 (Common Services and Abstractions): Layer 3 is the emphasis of EASIS work in software architecture, as shown in the Figure 9-1, the services are divided into three categories, the basic services, gateway services and ISS dependability software services (the blocks highlighted in blue in Figure 9-1).

- The first category comprises all basic software modules [BSW06], such as communication, diagnostics, calibration, network management and memory management, etc. These are gathered together as common services. All of these services are required to provide the upper functional layers with the platform they need for their functionality, e.g., the communication services provide abstract communication services and hide away all specifics arising from the physical communication systems chosen.
- The second category here is gateway services [GTD06], which support the applications distributed beyond network borders. The software services here, as well as ISS dependability services, are optional and configurable according to the requirement of the applications. The ISS gateway services consist of the services which are needed for safe and secure inter-domain as well as intra-domain communication. These services are not needed in non-gateway ECUs. In gateway ECUs some services like the Routing Service are required in any case, while others are optional depending on the required safety and security for the inter domain communication (e.g. Authentication Service and Firewall Service).
- The third category, as the focus of this dissertation, is the ISS dependability software services, which include different standardized software services to support safety relevant applications, such as Agreement protocol, Fault-tolerant Communication Services (as an add-on extension of common communication service), Resource Management and the Software Watchdog. It is notable that although the dependability software services here are categorized as parallel to the common software services, they are more like a configurable software service library compared with the basic software services, and make use of the API and services from the common software services.

Layer 2 (Microcontroller Abstraction and Device Driver): The Device Drivers and Microcontroller Abstraction Layer is an integration of hardware near software services. This layer provides the basic abstraction of the underlying hardware and the micro-controller and drivers of all devices included in the hardware platform.

The abstraction provides access to registers, I/O-ports, interrupts, etc. and hides away all specifics of the hardware and micro-controller. This interface does not provide any semantic information, meaning that data is handled exclusively as binary data, without any interpretation of the meaning of it. It also provides drivers such as network controllers, UARTs, digital I/O, AD/DA converters, TPU-functions (e.g. PWM) and so on. These device drivers provide some basic semantic information to the upper layers and may contain basic fault handling and other housekeeping functionality.

Layer 1 (Microcontroller hardware): Microcontroller hardware is layered at the bottom of this software architecture framework.

OS: In order to manage the execution of the software components included in the architecture, an operating system (OS), compatible to OSEK and AUTOSAR OS is used here. For some of the dependability services like Fault Management Framework, Software Watchdog and dynamic configuration services, extension to AUTOSAR OS will be needed. The placement of the OS between the L4 Interface Layer and L1 Microcontroller Hardware in the figure above is to indicate that it can be only accessed by layers between. In order to support dependability software services, an operating system with MPU (Memory Protection Unit) is about to be specified here in Subsection 9.3.2.2.

9.3 Concepts of the dependability software services

As stated in Subsection 1.1, two of the most distinguishing aspects of ISS are distribution of application across domain borders and integration of functions with different safety integrity levels.

Specific for these two challenges brought by ISS, four types of dependability software services are designed, prototyped and validated in this dissertation.

- Dependability services for ISS communication
- Dependability services for the integration of applications on one HW-platform
- Dependability services for fault treatment
- Dependability software services for the gateway services

9.3.1 Dependability services for ISS communication

The requirements of communication among ISS application components come from the design regarding data exchange and relationship between the functions. As depicted in Figure 7-7 about the ISS engineering process, if we take a close look at the mapping process from FAA to FDA as depicted in Figure 9-3, the purple lines denotes the communication between application software components in FAA-model. Such related application software components can be mapped to one ECU or different ECUs or even ECUs belong to different networks.

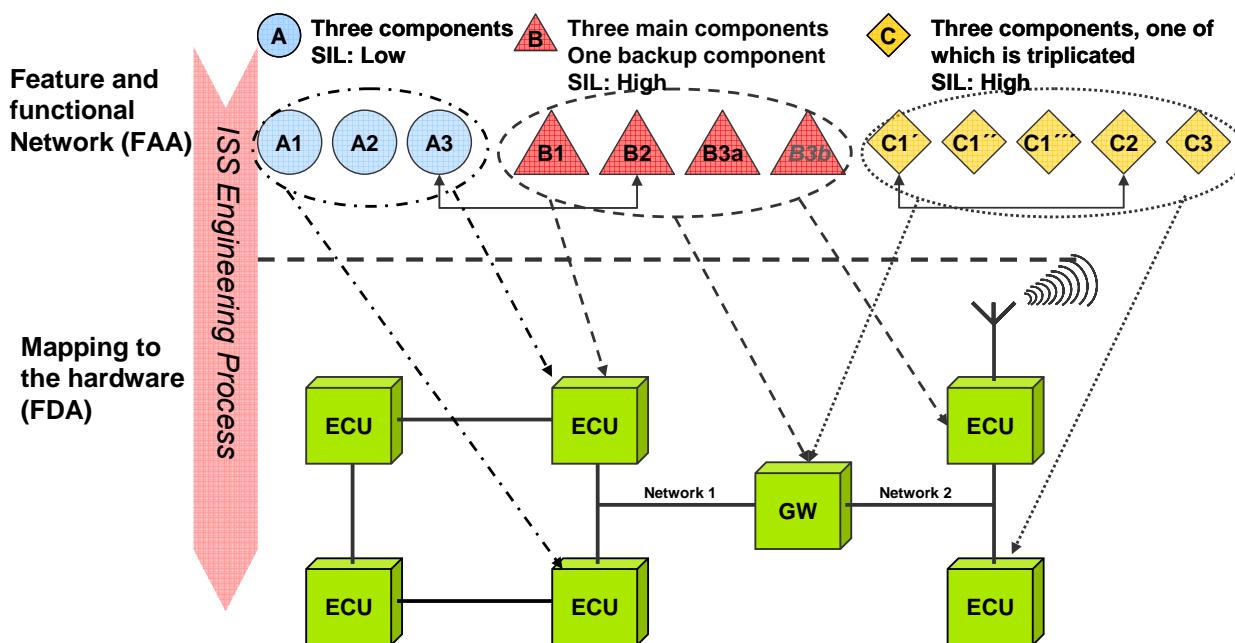


Figure 9-3: Mapping App.SW-Cs of different ASILs to the hardware topology

9.3.1.1 Categories of communication among ISS-application software components

As discussed in the mapping guidelines in Subsection 8.1.2 guideline 8, it is preferable to integrate only functions of the similar ASIL on one ECU. In practice, however, it is still possible to have

applications with high ASIL integrated with applications with low ASIL. In order to guarantee the availability of the applications with high ASIL, it is also possible to allocate the replica of applications with high ASIL to be integrated onto ECUs, which originally only host the applications of low ASIL. The communication between the software components in this scenario is depicted as following:

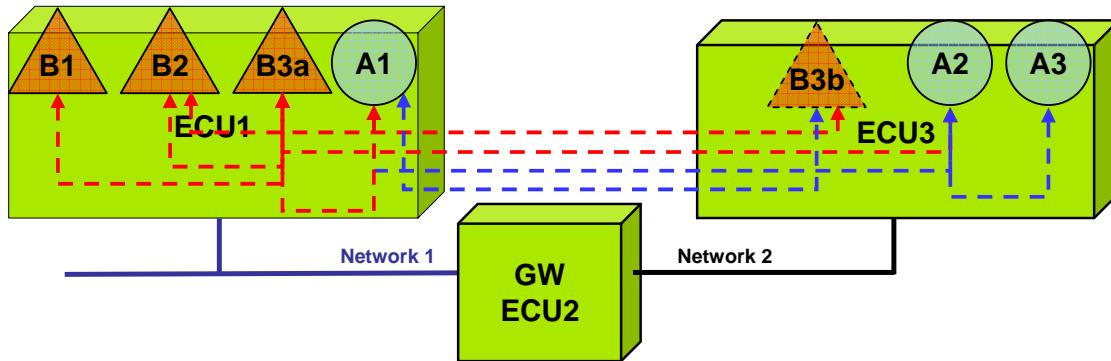


Figure 9-4: Example scenario of intra-ECU and inter-ECU communication

As denoted in Figure 9-4. ISS communication can be generally divided into intra-ECU communication denoted with the dashed red line, and inter-ECU communication, blue lines, e.g. ISS application B, highlighted in red, is made up of 3 software components B1, B2 and B3a, which are mapped to ECU1. The replica of B3a, B3b is mapped to the ECU3, where the app. SW-C A2 and A3 of lower ASIL are resident. App. SW-C A1 of low ASIL is also mapped to ECU1.

The communication between the app. SW-Cs can divided into following categories:

1. Mutual communication between App.SW-C of high ASIL with App.SW-C of high ASIL
2. Mutual communication between App.SW-C of high ASIL with App.SW-C of low ASIL
3. Mutual communication between App.SW-C of low ASIL with App.SW-C of low ASIL
4. Communication from App.SW-C of low ASIL to App.SW-C of high ASIL
5. Communication from App.SW-C of high ASIL to App.SW-C of low ASIL
6. Communication between the redundancies of App.SW-C of high ASIL

Among these 6 categories of communication:

- The category 1 refers to the intra-ECU communication between B1 and B3a or inter-ECU communication between B1 and B3b in Figure 8-4.
- The category 4 refers to the one way intra-ECU communication from A1 to B3a or inter-ECU communication from A1 to B3b, which is only a special case of communication category 2, where the App.SW-Cs of higher ASIL uses the information from App.SW-Cs of low ASIL.

- The category 6 is more or less a special case of category 1, since only the safety relevant app. SW-Cs can have replica. In order to reach a voted/agreed result among the replica, the replica should have the consistent inputs in the synchronized time window
An integrated solution for these will be discussed from Subsection 9.3.1.2 to Subsection 9.3.1.4, in which the discussed algorithms are mapped to the communication category 1, 2, 4 and 6.
- The category 3 and 5 are not safety relevant, since the consumer of the information is not of high safety integrity, thus they will not be discussed in details here.

9.3.1.2 Redundant communication service

As introduced in the Chapter 3, Redundancy is a common approach to improve the dependency. In case of communication, three types of redundancy may be used to achieve a fault tolerant communication:

- Information redundancy:** by information redundancy we mean redundancy of the complete information (e.g. redundant information from different memory areas by read/write) and abstract or footprint of the information (e.g. CRC).
Within a communication frame, Error detection mechanisms already present in most protocols are a common form of information redundancy. Completely replicating a signal within a frame is a possible solution for improving error detection or even performing error correction, but as it does not protect against loss of frame it is not as efficient as the other redundancy types.
- Time redundancy:** the signal is replicated in different frames or the same frame is sent multiple times.
- Structural redundancy:** a signal is transmitted on multiple physical communication channels.

In case of inter-ECU, the three categories of redundant communication are depicted in Figure 9-5.

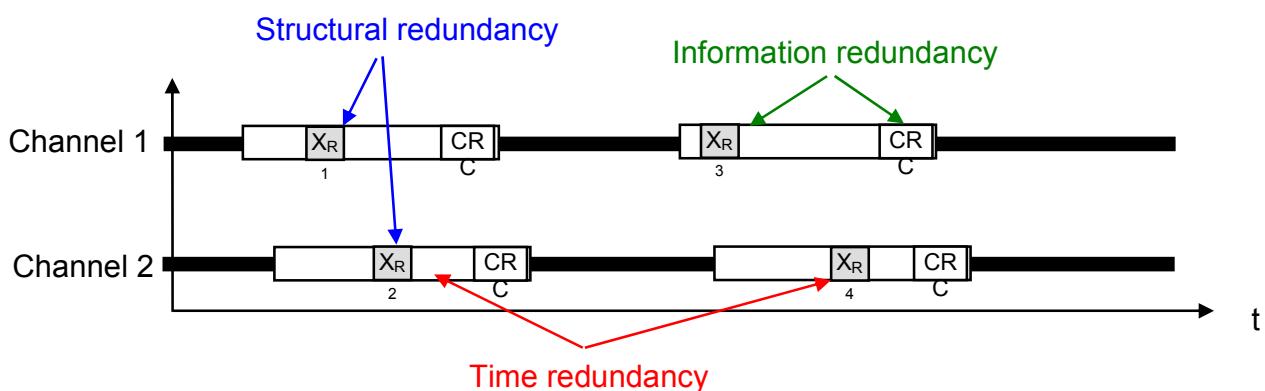


Figure 9-5: Communication redundancy mechanisms

The structure redundancy here fulfills the ASIL-D requirement of independent and redundant communication paths. In comparison, the information redundancy and time redundancy are cost effective, but require intelligent task scheduling, synchronization and plausibilisation of redundant information. While the information redundancy within one frame can be read almost once, the time redundancy can be a problem for the real-time control algorithms.

The redundant communication can be applied for the communication categories 1, 2 and 4 as specified in Subsection 9.3.1.1

9.3.1.3 End-to-end communication service with application CRC

As have discussed in the Subsection 8.2, various algorithms are applied for the fault detection of data transmission on the communication bus, but suppose that there should be some faults between the transceiver and application SW-C, e.g. memory error and application coincident fault in data read/write, such a fault can not be detected by the CRC in the communication protocol, since the protocol CRC only ensures the data integrity from bus transceiver of the sender to transceiver of the receiver. In Chapter 7 ISS EP Process Step 2.4, the mapping of system PFH to the PFH of FAA-components brings even higher safety integrity requirement of $1 \cdot 10^{-10}$, which can not be guaranteed with the bus protocol alone.

Figure 9-6 shows the end-to-end communication scenario between two ECUs and the possible communication fault model. Faults, occurred in the red high-lighted area can not be detected.

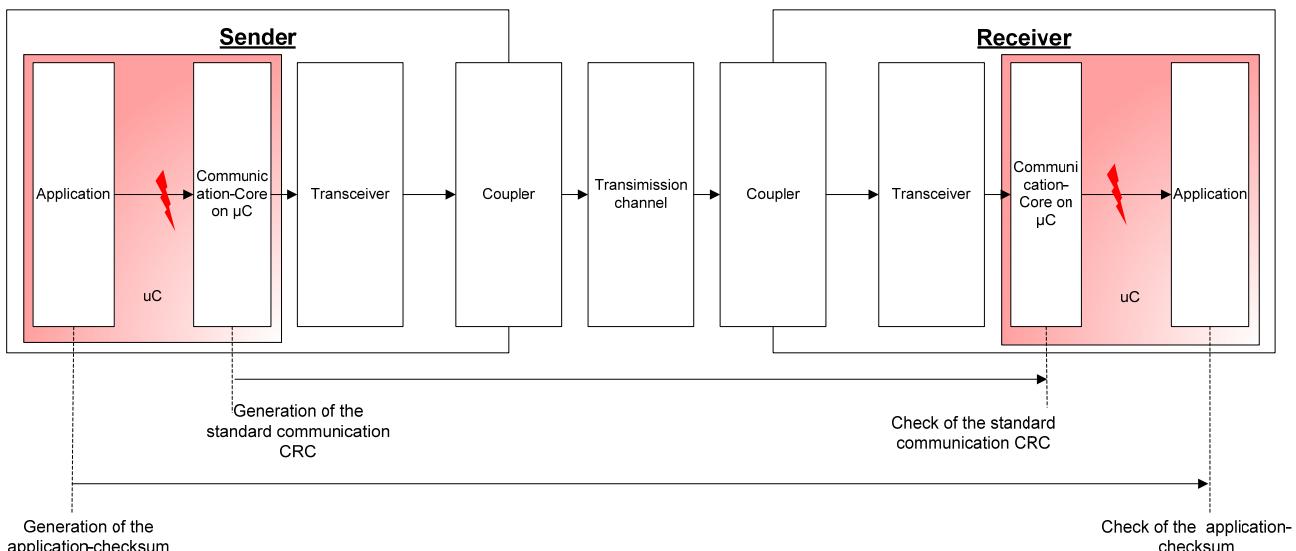


Figure 9-6: Reference model for end-to-end communication checksum

A detailed data flow from application SW-C to the transceiver is shown in Figure 9-7. On side of receiver as depicted in Figure 9-8, the same data transmission path in the inverse direction is vice-versa supposed.

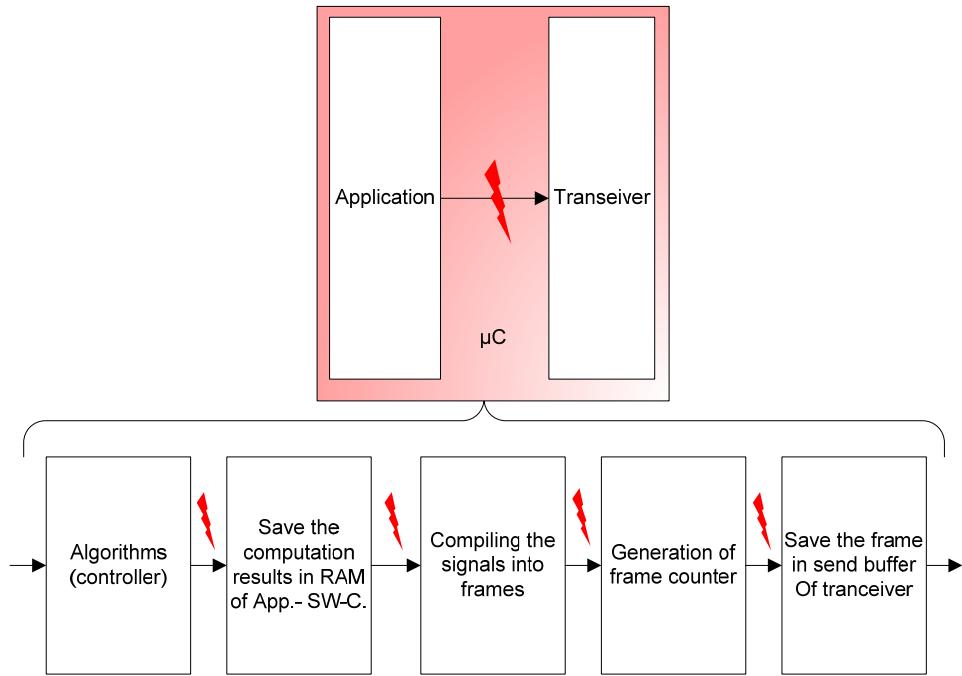


Figure 9-7: Data flow from application SW-C to transceiver (sender)

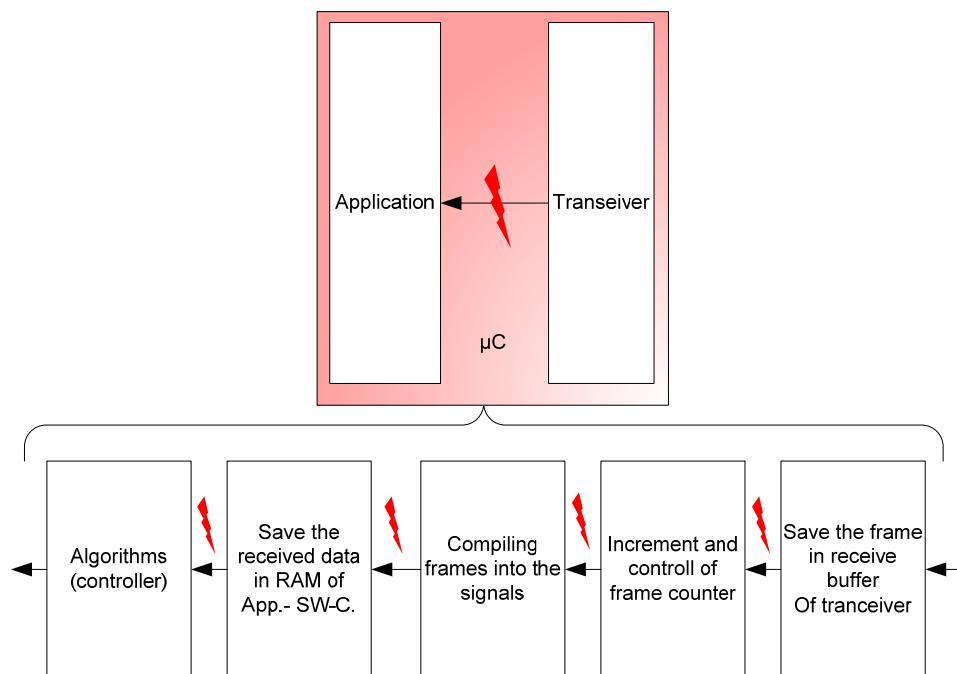


Figure 9-8: Data flow from transceiver to application SW-C (receiver)

A detailed list of fault types is given in Table 9-13, in which each chain of data flow of application SW-C to the transceiver can result in an error, which is not detectable with protocol CRC.

Ref.	Fault location	Fault type	Error
1.	RAM of App. SW-C	coincident bit-kipper	Data error
2.		DC fault model ³ of data and address	Data error
3.		Dynamic cross-over of memory cell	Data error
4.		Faulty address by read/write operation	Data error
5.	Program counter, Stack pointer	DC fault model*	Data error
6.	Program control flow	Not defined	Data error
7.	μ C interrupt	Cross-over von Interrupts	Data error
8.	CPU Address	Not defined	Data error

Table 9-13: Fault types of date flow from application SW-C to transceiver

Thus it is reasonable to put an application CRC in the earliest chain of data flow of the sender (or the last chain on side of the receiver), as shown in the following Figure 9-9, so that faults in the following chains of data transmission can be also detected with application CRC.

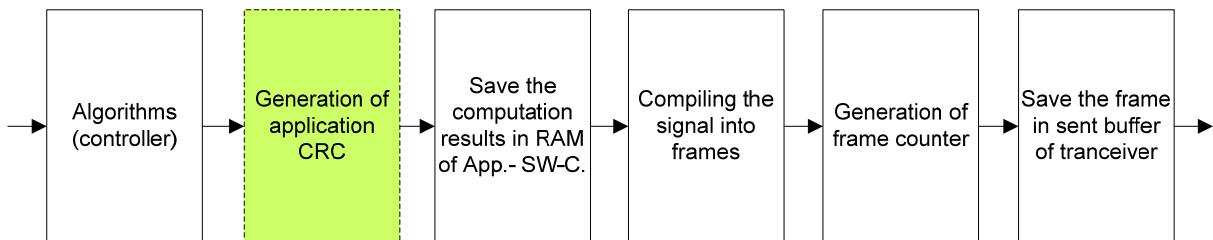


Figure 9-9: Date flow with application CRC on the sender

When decision is made to use application CRC, it is still needed to decide, which CRC-algorithm should be applied for the implementation of the safety relevant application.

Let us go back to the dummy ISS application SafeSteering as discussed in Figure 8-13. In SafeSteering, the additional wheel angle superposition is sent from BMU to SMU, and the both applications are ASIL-D with PFH requirement of $< 10^{-8}$. As discussed in ISS EP Step 2.4 Figure 7-7 about PFH proportion of communication system to the system PHF, the communication channel between the two systems, as a system component, should be implemented with a PFH of $\leq 1 \times 10^{-8} \times 1\% = 1 \times 10^{-10} / h$ (even higher than ASIL-D). That is to say, the possibility of a fault in the communication channel, which could result in a system wide safety relevant failure, should be

³ DC (direct current) fault model includes: stuck-at faults, stuck-open, open or high impedance outputs and short circuits [IEC02]

smaller than 1×10^{-10} per hour. Based on the safety integrity of the application, a few questions should be clarified before the design:

- What kind of system fault tolerant behavior is required here, fail-safe or fail-operational?
- Is it enough to detect the communication fault (fail-safe) or do we need to correct it (fail-operational)?
- How sensible is the algorithms of control loop as to an undetected faulty input data? It is common that not every faulty transmitted message could result in a faulty control output (also called process safety).

In order to answer these questions, a system wide consideration should be carried out, e.g. the plausibility monitoring of input values and the controllability of the driver in case of faulty output should be considered in the tolerance of communication fault as well.

In SafeSteering, if CAN is used for the communication between Steering Management Unit and Braking Management Unit, additional dependability software services are required for the ASIL-D signals. Take the wheel steering position as an example: let us assume the pay-load of the steering angle as p bits, which are sent out every t_s s, while the total data transfer rate is v bits/s.

In order to describe the quality of physical transmission, term of Bit Error Rate (possibility of corrupted received bits) is introduced. For the automotive communication system, an assumption is made that the Bit Error Rate varies from 10^{-4} in case of an aggressive environment to 10^{-6} in the case of a benign environment. According to an experience research about Bit Error Rate of CAN [BER04], even under an aggressive environment the Bit Error Rate of CAN is only $2.6 \times 10^{-7}/h$.

Three different integrity classes (I1, I2 and I3) are assigned in the industry norm [DIN93] to the communication path, in which I1 has a hamming-distance of at least 2, while I2 and I3 has a hamming-distance of at least 4. If we choose a relative conservative value of the Bit Error Rate of 1×10^{-4} , as the same assumption from [DIN93], the probability of undetected error in the communication channel (denoted as R) is 10^{-6} by I1, 10^{-10} by I2 and 10^{-14} by I3 ([DIN93] diagram 1 about data integrity class). Here it is to mention that, only the undetected communication error is dangerous for the system (as dangerous failure undetected by the diagnostic test λ_{DD}).

As depicted in Figure 2-3 of Subsection 2.1, the latency time of fault tolerance $t_4 - t_1$ should be smaller than $t_3 - t_1$, during this latency time, the number of faulty messages (process safety) we can tolerate can be calculated as

$$m = \left\lfloor \frac{t_6 - t_4}{t_s} \right\rfloor$$

That is to say, only if $m+1$ successively faulty messages are undetected received and used by the control algorithms, a system wide dangerous failure could happen. We assume the worst-case here that each system failure will result in an uncontrollable safety relevant failure regardless of the driving situation and driver controllability.

The possibility of a system safety relevant failure resulted from the communication failure can be calculated as

$$P_{com} = R^m = R^{\left\lfloor \frac{t_6 - t_4}{t_s} \right\rfloor}$$

In one hour, there are n bits sent to the receiver ($n=3.6*10^3*v$), let us take another worst-case assumption that each undetected faulty bit can result in a undetected faulty message. Since only m successive undetected message can result in a system failure, the number of system failure in one hour can be calculated as

$$FH = Data_transferate * 1_hour * possibility_of_system_failure \\ = n * P_{com} = 3.6 * 10^3 * v * R^{\left\lfloor \frac{t_6 - t_4}{t_s} \right\rfloor}$$

, which should be smaller than the Dangerous Failure per Hour (PFH) required by AISL-D of $1*10^{-10}$. A further research of the formal above shows that the parameters t_s , t_6 , and t_4 are system and application specific and can be less influenced by the application CRC. The decision of the CRC algorithms, however, influences the value of R directly.

In the following the whole approach to determine the CRC-algorithms is demonstrated with the SafeSteering, in which the parameter used are exemplary but not directly from the praxis.

Let us take high-speed CAN with 500 Kbit/s as v , system reconfiguration time t_6 as 10 ms, t_4 as 7 ms and steering angle superposition information is sent every 1 ms, during the reconfiguration time

$t_6 - t_4$, we can tolerate $\left\lfloor \frac{t_6 - t_4}{t_s} \right\rfloor = (10 - 7)/1 = 3$ faulty undetected messages, thus

$$FH = 3.6 * 10^3 * v * R^{\left\lfloor \frac{t_6 - t_4}{t_s} \right\rfloor} = 3.6 * 10^6 * 5 * 10^5 * R^3 \\ \text{and } FH < 1*10^{-10}$$

We could get that R should be smaller than $3.8*10^{-8}$. Again according to [DIN93] diagram 1, the integrity class 2 and class 3 should be applied, in which a hamming distance of at least 4 is required.

According to the Figure 2 and Table 3 in the paper of Koppman [Kop04], a hamming distance of 4 can be reached by 8 bits CRC as CRC8 and CRC0x97, 7 bits CRC as 0x5B and even 6 bits CRC as 0x2C, etc. In the embedded system of automotive industry, the CRC-8 is a quite commonly applied algorithm, thus the same algorithm can be applied for the application CRC with reuse of legacy codes.

9.3.1.4 Agreement Protocol

As specified in Chapter 6 about fault hypothesis and fault types for ISS, the Byzantine General problem are introduced.

A common approach to implement fault-tolerance in the safety relevant systems is the use of structural redundancies, defined in [Ech90] as the expansion of the system by additional components (software or hardware) to realize the same function redundantly. The structural redundancies within an electronic system however, introduce some problems themselves.

As discussed in [Bes02] and [Ech90], different sources of undesirable non-determinism can be identified. The simplest non-determinism is in value domain. For example, while measuring the same dimension using redundant sensors, the measured values, which theoretically should be identical, may exhibit slight deviations due to the physical nature of the sensors and are therefore

non-deterministic. The use of these values leads to the computation of non-deterministic results, which is not acceptable for safety-related applications. Inconsistency in the value domain for structural redundancy can be solved by fault masking, which assumes the presence of a k -out-of- n system (see Figure 9-10). This consists of n redundant components, at least k of which must be non-faulty to ensure fault-tolerance.

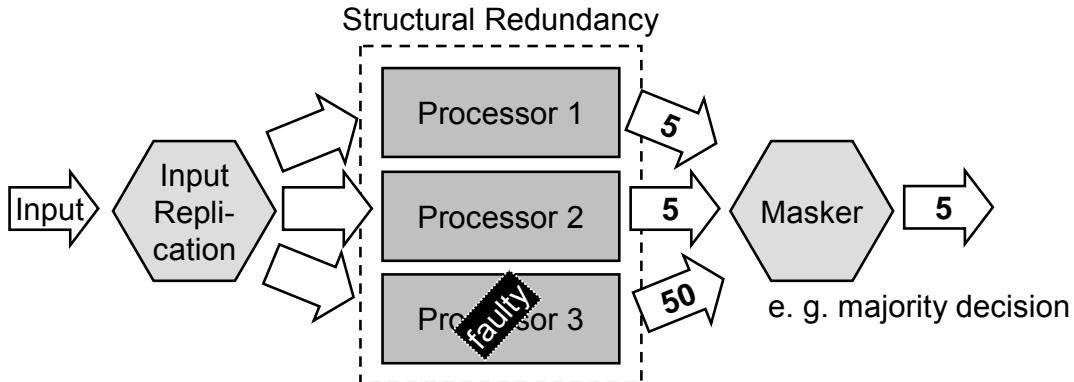


Figure 9-10: Fault masking 2-out-of-3 system [Ech90]

Given the high requirements for reliability in some applications, the masker, which constitutes a single point of failure, should also be replicated. Since additional communication is required to reach a consistent decision among maskers, the overall messages overhead is increased [CHS06].

The other form of non-determinism affects the time-domain and arises in the presence of a high latency jitter in the communication system, so that neither the arrival time of redundant messages nor their sequence in time can be predicted accurately. Such non-determinism in the time domain can be solved by the synchronization of data transmission and simultaneous data processing on redundant nodes. The introduction of a time-triggered in-vehicle communication system such as FlexRay and a predictable ECU operating system to guarantee the time constraints, simplifies the synchronization effort significantly.

Another form of non-determinism as the worst of all, is asymmetric fault (also known as the Byzantine fault, as discussed in [Lam82], Subsection 6.2) can also occur due to software and hardware coincident fault,

To ensure fault-tolerant communication even with Byzantine Faults, Lamport et al. provide different algorithms to solve the problem in [Lam80], so that non-faulty components can reach agreement on a value regardless of faults. These algorithms allow the non-faulty nodes to compute the same *interactive consistency vector* (*ICV*) containing the private value of each non-faulty node. Once interactive consistency is achieved, each non-faulty node can apply an averaging or filtering function to the interactive consistency vector, according to the needs of the application. In the following, a brief introduction to the different variants of Agreement Protocols as well as the assessment to the dependability requirements in the automotive industry is given:

Oral Messages Protocol

The notion of oral messages was introduced together with the Byzantine Generals problem ([EiK06], [Lam80], [FTD06]) and the definition of an oral message is embodied in the following assumptions with regard to the communication system:

- Every sent message is correctly received.
- The sender of a message is known to the receiver.
- The absence of a message can be detected.

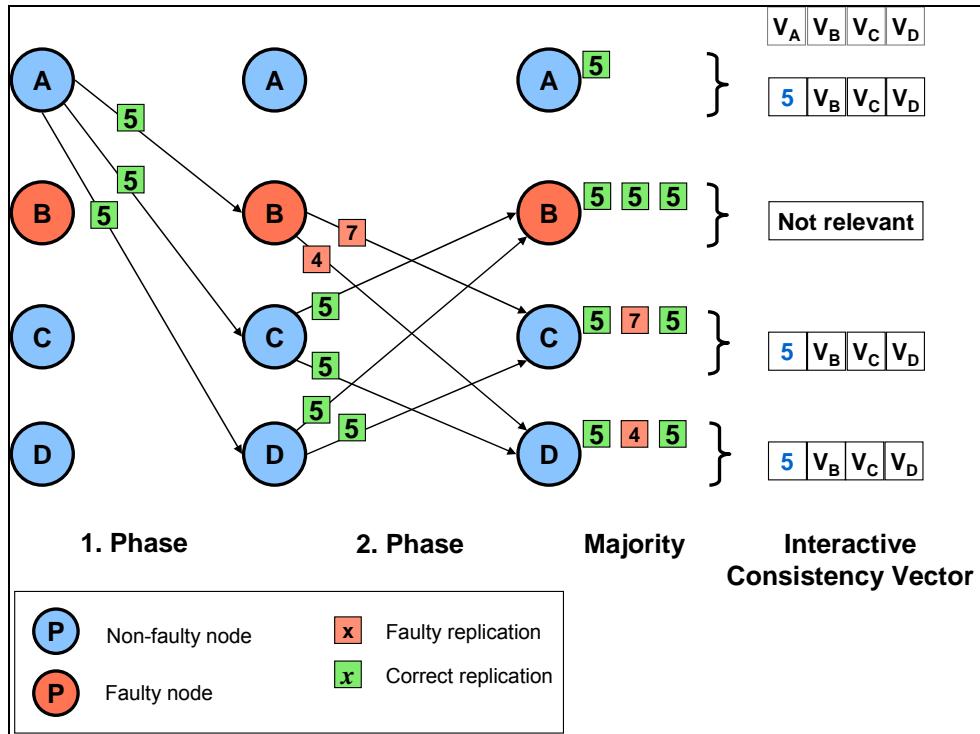
As depicted in Figure 9-11:

- P is the set of all nodes.
- V is the set of all values.
- A k -level scenario σ is a mapping from a non-empty string of length $\leq k+1$ over P to V , thus summarizing the outcome of a k -phase exchange of messages. For example, let $w = p_1 p_2 \dots p_r$. $\sigma(w) = v$ is then interpreted as the value p_2 tells p_1 that p_4 told $p_3 \dots$ that p_r told p_{r-1} is p_r 's private value. $\sigma(p)$ simply designates the private value of node p and σ_p is the restriction of σ to the messages received by p in a scenario (i.e. only for $w = pw'$)

In the following, the algorithm $\text{OM}(m, n, \sigma)$ is described in terms of p 's computation of the element of the Interactive Consistency Vector ICV corresponding to each node q , for a given scenario σ , an arbitrary number of faulty nodes $m \geq 0$ and a total number of nodes $n \geq 3m+1$:

- (1) If for some subset Q of P of size $> (n+m)/2$ and some value v , $\sigma_p(pwq) = v$ for each string w over Q of length $\leq m$, p records v .
- (2) Otherwise, q must be faulty. The algorithm $\text{OM}(m-1, n-1, \hat{\sigma})$ is then recursively applied with P replaced by $P - \{q\}$, and $\hat{\sigma}$ defined by $\hat{\sigma}(pw) = \sigma(pwq)$ for each string w of length $\leq m$ over $P - \{q\}$. This recursive call delivers an ICV with $n-1$ elements, since the node q is excluded. The i -th element corresponds to the value all non faulty nodes agree it has been received by the i -th node p' from q during the first phase (i.e. $\hat{\sigma}(p')$ which is in fact equal to $\sigma(p'q)$). If at least $\lfloor (n+m)/2 \rfloor$ of the $n-1$ elements agree, then p records the common value. Otherwise, p records *nil*.

The correctness of this algorithm as well as the necessity of $n \geq 3m+1$ is proved in [Lam80].


 Figure 9-11: OM algorithm with $n = 4$ and $m = 1$

Signed Messages Protocol

Since oral messages can be easily forged by faulty nodes before being forwarded, reaching agreement among non-faulty nodes with the help of the OM algorithm requires at least $n = 3m + 1$ nodes to tolerate m faulty ones and hence induces a significant communication overhead. By using signed messages instead of oral messages, agreement can be reached for any number $n > m$ nodes and also requires $m + 1$ communications phases.

As opposed to oral messages, a signed message from a node p including some data item d cannot be forged, as this message holds an authenticator $A_p[d]$ from p , allowing each receiver to check the message integrity as well as the identity of the original sender. Since this authenticator cannot be falsified, faulty nodes are no longer able to alter the content of a message before forwarding it without being found out. Moreover, forwarded messages are additionally signed by the forwarding node.

Let $v = \sigma(p)$ for a node p (see the subsection above about Oral Message Protocol for the definition of σ). p communicates this value to another node r by sending the message consisting of the triple (p, a, v) , where $a = A_p[v]$. After receiving the message, r checks whether $a = A_p[v]$. If the test is successful, r takes v as the value of $\sigma(p)$ (i.e. $\sigma(rp) = v$) and forwards the message $(r, A_r[(p, a, v)], (p, a, v))$ to all other nodes except p ; otherwise $\sigma(rp) = \text{nil}$.

More generally, if r receives a message of the form $(p_1, a_1(p_2, a_2, \dots, (p_k, a_k, v), \dots))$, where $a_k = A_{p_k}[v]$ and for $1 \leq i \leq k - 1$, $a_i = A_{p_i}[(p_{i+1}, a_{i+1}, \dots, (p_k, a_k, v))]$, then $\sigma(rp_1 \dots p_k) = v$ and r forwards the message after signing it to every other node different than p_1, p_2, \dots and p_k ; otherwise, $\sigma(rp_1 \dots p_k) = \text{nil}$.

In the following, the signed messages algorithm SM (m, n, σ) is described in terms of the value a non-faulty node p records for a given node q and a scenario σ . m designates the number of faulty nodes ($m \geq 0$) and n the total number of nodes ($n > m$).

Let S_{pq} be the set of all non-nil values $\sigma_p(pwq)$, where w ranges over strings of distinct elements with length $\leq m$ over $P - \{p, q\}$. If S_{pq} has exactly one element v , p records v for q in its ICV; otherwise, q must be faulty and p records *nil*.

As proved in [Lam80], this algorithm allows the non-faulty nodes to compute the same ICV containing the private value of each non-faulty node and *nil* for each faulty node. Although $n > m$ nodes are sufficient to allow non-faulty nodes to reach agreement in the presence of m faulty ones, it makes more sense to use $n \geq 2m + 1$ nodes, since non-faulty nodes must be in majority. This would enable a downstream masker using a majority decision to vote for the correct result.

As depicted in Figure 9-12, the ICV of each node is computed in the distribution protocol during the communication phases. Before the first phase, each node initializes its ICV to *nil* values except for its corresponding element where the node's private value is recorded. The vector is then actualized upon each communication phase as follows. When a message $(p_i, a_i(p_j, a_j \dots (p_k, a_k, v) \dots))$ is received by a node p and turns out to be valid, then

- If $ICV_p[k] = nil$, then the value v is recorded for the k -th node, so that $ICV_p[k] = v$.
- If $ICV_p[k] \neq nil$ and $ICV_p[k] \neq v$, then a replication fault obviously occurred in the node p_k and $ICV_p[k]$ is set to *faulty*, so that any further values for p_k are ignored. Nevertheless, valid messages are always forwarded independently of the value of $ICV_p[k]$.

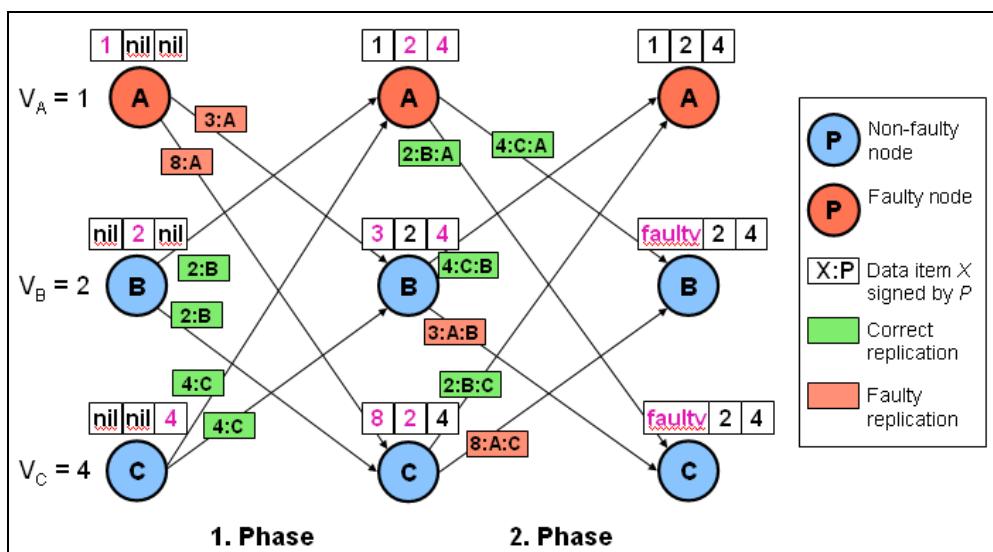


Figure 9-12: Time diagram of the Signed Message Protocol with 3 nodes

Pendulum Protocol

Independently of whether faults actually occur, the previously presented protocols always induce a worst-case communication overhead, since an interactive consistency vector is computed for all nodes before a final result is decided. The *pendulum protocol*, which is introduced in [Ech90], aims at reducing the communication overhead for the faultless scenario at the expense of fault detection by computing a single agreed value. The name of the protocol comes from its algorithm which executes in a way similar to a pendulum stroke over the nodes.

The pendulum consists of a *main protocol*, which is sufficient in the faultless case, and an *auxiliary protocol* applied to deal with faults when these occur. Moreover, it requires the use of some *distance decision* (i.e. interval decision or sphere decision) and thus the definition of δ as depicted in Figure 9-13.

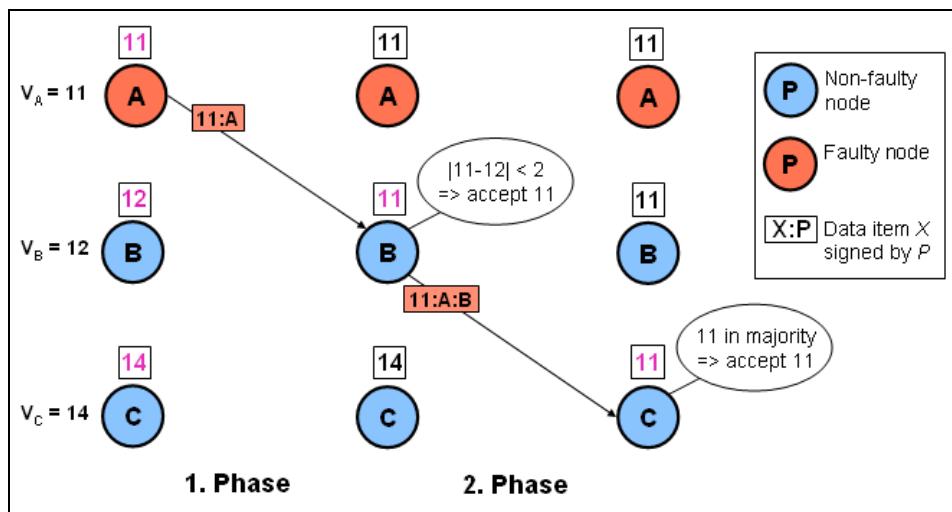


Figure 9-13: Pendulum Protocol for 3 nodes with $\delta = 2$

Assessment of the agreement protocol variants

The following criteria were applied during the assessment of these three variants of protocols:

- Predictability: the algorithm behavior does not depend on run time events (e.g. the occurrence of faults).
- Fault-detection: the algorithm includes mechanisms that enable the detection of components providing faulty values.
- Fault-tolerance: agreement can be reached despite the presence of faults.
- Number of redundancies: the minimal number of redundancies required to tolerate m faulty processing units.
- Minimal communication overhead: the minimal number of messages required to reach agreement among n processing units, e.g. in the absence of faults.
- Maximum communication overhead: the maximal number of messages required to reach agreement among n processing units, e.g. in the presence of faults.

- Number of communication phases: the number of communication phases required to tolerate m faulty processing units.

These criteria can be classified with respect to their priorities, as depicted in Table 9-14, which categorizes the properties of the agreement protocols in three levels, Priority 1, Priority 2 and Priority 3.

Priority 1	Priority 2	Priority 3
<ul style="list-style-type: none"> ▪ Predictability ▪ Fault tolerance ▪ Fault detection 	Number of redundancies	<ul style="list-style-type: none"> ▪ Minimal com. overhead ▪ Maximum com. overhead ▪ Number of com. phases

Table 9-14: Priorities of Criteria for the assessment of agreement protocols

Based on these criteria, the three agreement protocols are compared in Table 9-15, where n is the total number of processing units and m the number of faulty units to be tolerated. The mathematical derivation these results can be found in Appendix. 2

	OM Algorithm	SM Algorithm	Pendulum Protocol
Predictability	yes	yes	no
Fault tolerance	yes	yes	yes
Fault detection	yes	yes	no
Number of redundancies	$3m+1$	$2m+1$	$2m+1$
Minimal com. overhead	$\frac{n \cdot (n-1) \cdot ((n-1)^{m+1} - 1)}{n-2}$	$\frac{n!}{(n-2)!} + \dots + \frac{n!}{(n-m-2)!}$	$n-1$
Maximal com. overhead	$\frac{n \cdot (n-1) \cdot ((n-1)^{m+1} - 1)}{n-2}$	$\frac{n!}{(n-2)!} + \dots + \frac{n!}{(n-m-2)!}$	variable
Number of com. phases	$m+1$	$m+1$	variable

Table 9-15: Comparison of three agreement protocols [Appendix 2]

From the results of this comparison, the pendulum protocol does not seem to be adequate even if it provides the lowest communication overhead in the faultless scenario. In fact, besides failing to fulfill the predictability requirement, it does not provide 100% fault detection and could even lead to the agreement on a faulty value. Moreover, the pendulum protocol is limited to applications supporting only an interval or a sphere decision, which makes it less qualified for a standard service implementation.

Both the OM and the SM algorithms fulfill the most relevant requirements. Although the signing of messages induces an additional computational overhead, the SM protocol is most appropriate in the scope of this work, since it can guarantee fault-tolerance with less redundancies and a lower communication overhead than the OM protocol.

Based on the safety requirements in automotive industry, the Agreement Protocol service is embedded in the dependable software architecture specified in Subsection 9.2. As depicted in Figure 9-14, the application using the AP service must provide an input value as the private value of the node, the number of phases (depending on the number of faults to be tolerated), the tolerance band and additional node information required for the signature mechanism. The data exchanges are carried out using a fault-tolerant communication service via FlexRay.

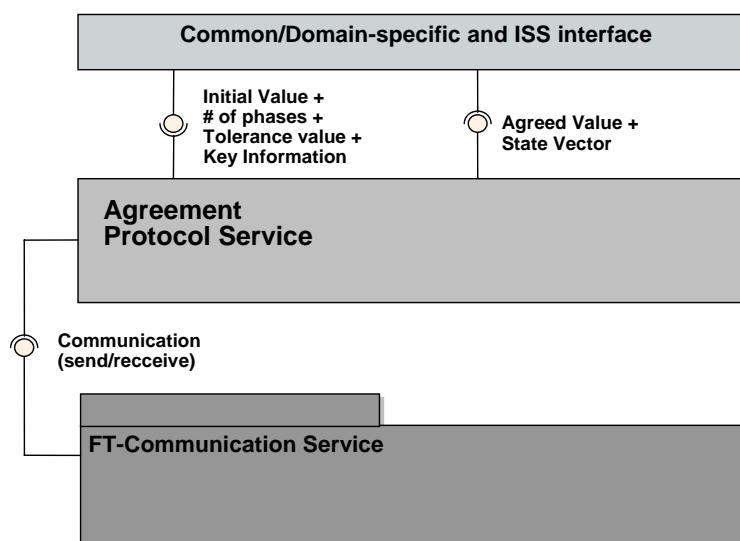


Figure 9-14: Interface to fault-tolerant communication and application

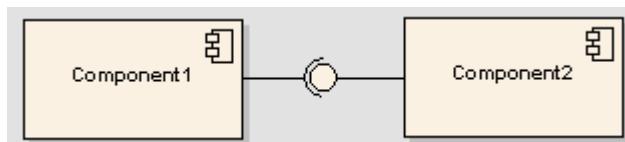


Figure 9-15: Assembly symbol in UML with Enterprise Architect

The “assembly” symbol of UML 2.0 used in Figure 9-14 is a connector between two components, which denotes that one component provides the services that another component requires. As shown in Figure 9-15 the assembly bridges a component’s required interface (Component1) with the provided interface of another component (Component2).

9.3.1.5 Conclusion of ISS communication

As discussed in the subsections above, the ISS communication is characterized with the high dependability requirement (ASIL-D communication with PFH of 10^{-10}). Fault tolerance in each chain of the communication channel even with the presence of Byzantine Faults is required for the future ISS applications. The approaches presented in this subsection focus on the fault-tolerant communication services, which are integrated in the software platform as dependability software

services. In the praxis, the real implementation should be tailored, configured and supplemented with application specific services such as identifier for the communication objects, aliveness counter, error detection code, acknowledgement, etc. Under the requirement that safety integrity level from the safety analysis should be met, a balanced communication concept considering system complexity can be chosen.

9.3.2 Dependability services for the integration of applications on one HW-platform

As have mentioned in Subsection 9.1, there is the trend of increasing integration density of applications on one HW-platform, in which the main challenges are as following:

1. Managing the complexity in the integration process of various functions on the same ECU during the system development
2. Increasing the system overall dependability by the partitioning of applications, which have different safety integrity levels

The first challenge is mainly considered in Chapter 7 with the help of ISS Engineering Process, while the second topic should be solved in an integrated manner with a dependable architecture framework.

9.3.2.1 Challenges and consideration from an integrated manner

As specified and required in [ISO26] Baseline 9, Chapter 6 Software design 6.4.10 and 6.4.11, freedom of interference by software partitioning is one of the most important technologies for the integration of applications with shared resources. The shared resources on ECU refer to computation time, memory and communication. While the dependable communication is discussed in Subsection 9.3.1, the partitioning topic here focuses in the computation time and memory.

The term of partitioning is meaningful when two different entities can influence each other by means of interface or shared resource. Generally speaking, partitioning is a fault containment technique which consists in precluding a failure in a partition to propagate (to control the additional hazard) and to cause other partitions to fail, which can be divided into hardware partitioning and software partitioning.

HW-partitioning is discussed in Chapter 8 by means of guideline to map applications to the HW-Architecture and hardware design. Partitioning, when applied to software, the intent is to control the additional hazard that is created when a software partition shares its or part of its resources with other partitions (shared resource of memory, computation time and peripheral access, etc.). Partitioning with clear interface definition and layered architecture is the topic of software design, while partitioning in this chapter refers to the software partitioning with shared resource and software partitioning is not intended to protect against failure of each software partition but against their propagation.

Since we are talking about partitioning, it is necessary to define the software framework to be partitioned with the help of dependable ECU software architecture as defined in Subsection 9.2.

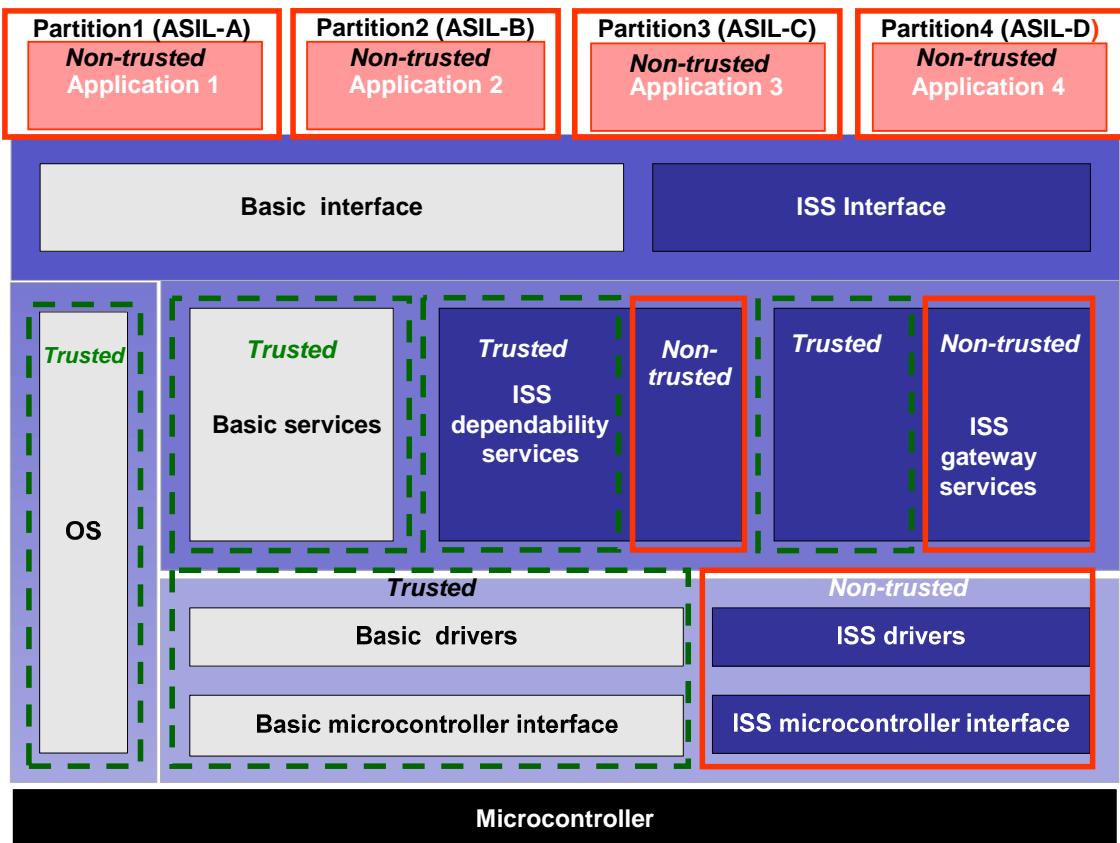


Figure 9-16: Software partitioning in the dependable software architecture

As depicted in Figure 9-16, on top of the dependable software architecture various application software components with different safety integrity levels can be integrated. As mentioned in Subsection 9.3.1.1, it can still happen, that ASIL-B functions are integrated with the ASIL-D functions if they all belong to the same ASIL-D safety application, e.g. diagnostic function in the SafeSteering application mentioned in Subsection 8.4.1. In this case, the functions with different ASILs should be partitioned to suppress the mutual influence. For same reason, the underlying software framework belong the interface layer (below EASIS SW-layer L4) could come from difference sources (different suppliers) as well. They are not all standardized and developed to the ASIL-D requirement, thus they should be partitioned to the trusted and non-trusted software modules. As the emphasis of partition, the App.SW-Cs are generally considered as non-trusted. Basic software services / drivers, OS, which have been used in many serial projects, are considered as trusted object. Part of fully validated ISS-dependability and gateway services can be identified as trusted-objects as well. Since the regulation of partition are completely or partly deactivated for trusted object. The trusted objects, which have a higher flexibility than non-trusted objects, can be the more dangerous for the system in case of error.

Software partitioning in this subsection encompasses space partitioning and time partitioning:

- Space partitioning addresses unauthorized data access and illegitimate commands of peripherals assigned to other partitions, which is discussed in detailed in Subsection 9.3.2.2 [ISO26].
- Time partitioning that focuses on disturbance of the timing of events in other partitions (scheduling, order of execution, etc.) is addressed as in Subsection 9.3.2.3.

9.3.2.2 Space partitioning with protection services

When talking about space partitioning, it is necessary to define the objects on the ECU, which are to be partitioned.

Following the ECU software development process, the next mapping step after Figure 9-3 is the mapping of application runnables to the task on the ECU as depicted in Figure 9-17, where runnables from one App.SW-C can be mapped to different ECUs, runnables from different application SW-Cs can be mapped to the same task as well. In order to save resource esp. computation time, state of the art approach is to pack the functions (runnables are defined in Figure 9-17 as the atomic code sequences of App.SW-C), which are executed to the same time cycle, together to one frame as a task. In an embedded OS, which is both compatible with time-triggered and event-triggered scheduling, tasks could have different priorities, in which task with lower priorities can be preempted by task of higher priorities. Usually the most important tasks are assigned with the highest priority and executed with the highest frequency supported by the microcontroller. Tasks with lower priority, which do not have the hard real-time requirements, can be preempted to guarantee that, sufficient resource is available for tasks of higher priority.

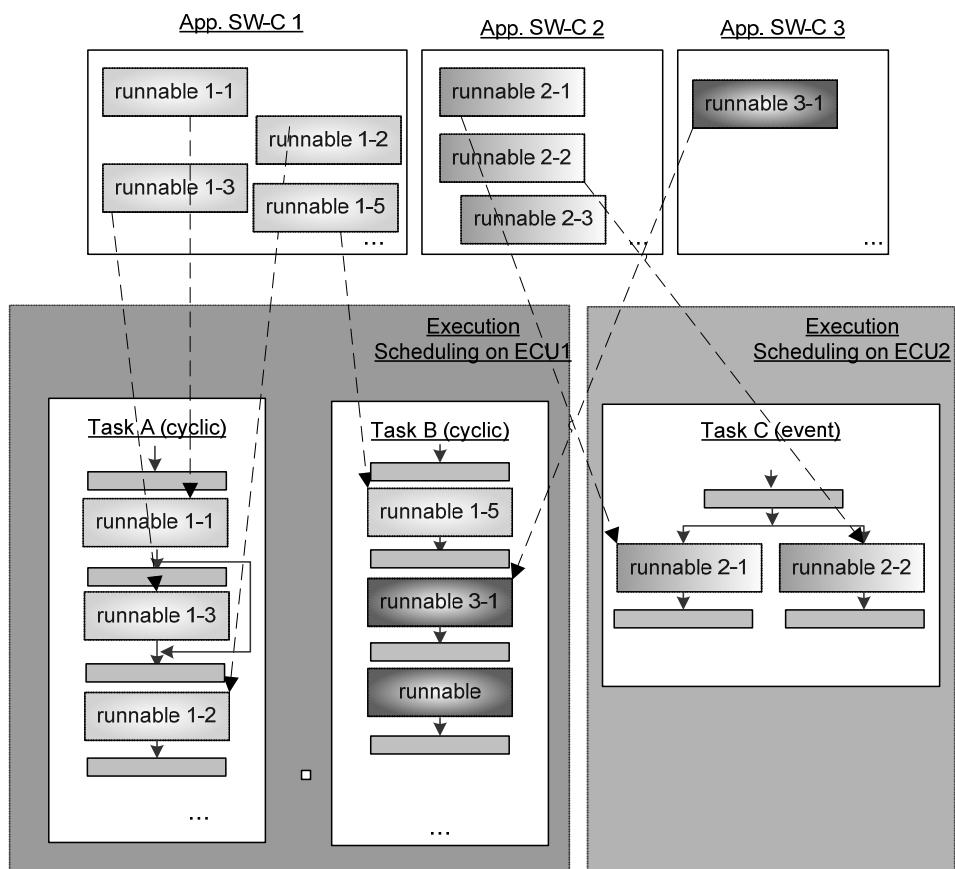


Figure 9-17: Mapping of application to ECUs and runnables to tasks

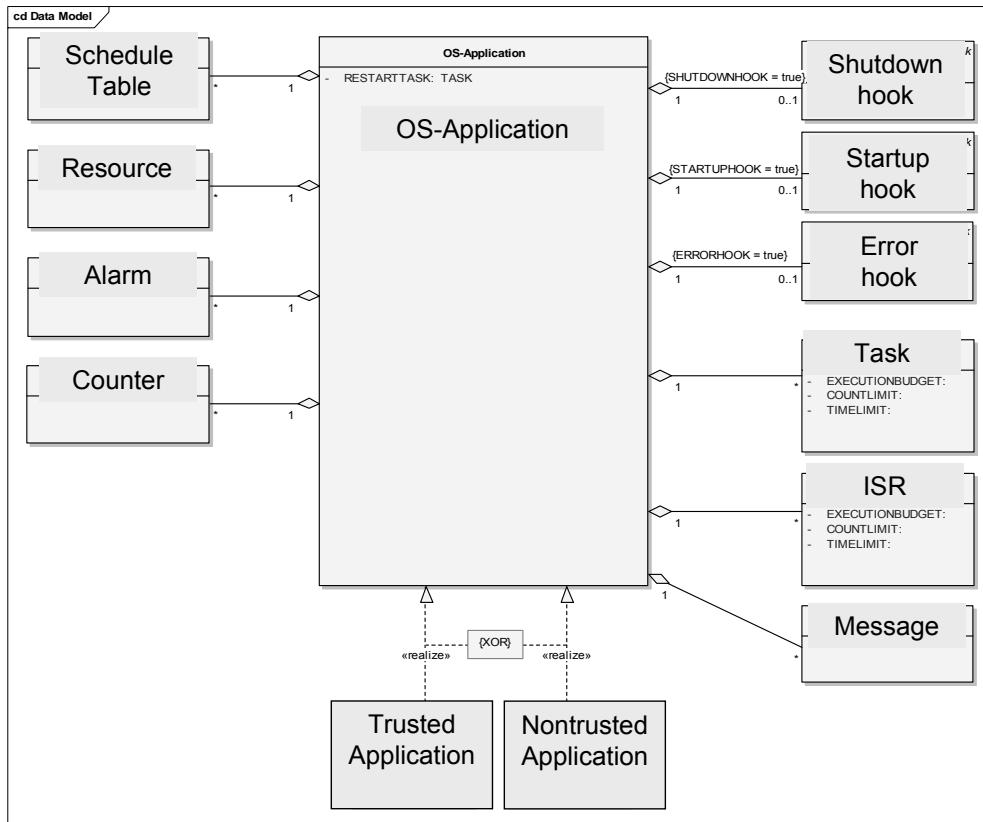


Figure 9-18: UML-model of objects in operating system

From view of operating system, all application SW-Cs, basic SW-Ms, dependability SW-Ms, etc. are implemented in OS-objects as depicted in as in Figure 9-18, such as schedule, resource, alarm, counter, hook, task, ISR and message etc. The most frequently used OS-objects are task, alarm, counter and hook. The space partitioning should be built up based on these OS-objects with help of system design, MMU/MPU and OS support. The OS-objects shown in Figure 9-18 can be restructured into two classes [AOS06]:

1. Trusted OS-objects are allowed to run with monitoring or protection features disabled at runtime. They may have unrestricted access to memory, the OS API, and need not have their timing behavior enforced at runtime. They are allowed to run in privileged mode when supported by the processor. In this sense, trusted OS-Applications are easy to use but dangerous.
2. Non-Trusted OS-objects are not allowed to run with monitoring or protection features disabled at runtime. They have restricted access to memory, restricted access to the OS API and have their timing behavior enforced at runtime. They are not allowed to run in privileged mode. Non-trusted applications are secured applications that they can not easily disturb other applications.

Especially for the non-trusted OS-objects, the first step of space partitioning is that they should be structured already in the design phase properly. This means, that they must be structured according to their function and not e. g. according to their time-frequency. Runnables with high priorities, which do not belong to the same function, should be structured to different tasks. If the given functionality of OS-objects regarding data encapsulation is not used, the problem of unintended writing in memory can of course not be solved. If it is used, an MMU enables protection

even in case of programming errors that an application tries to write to the memory region of another application. When a task intends to access a memory area by write operation, by comparison of the task ID with a preconfigured task/memory allocation table, OS could trigger an error hook if the task is not authorized to perform the operation. The memory partitioning could consider the following rules:

- Each task is assigned with a memory area for all possible operation.
- For tasks, which belong to the same application, can be assigned with the application wide shared memory area.
- Eventually part of memory area can be reserved for all applications for a free access.
- As the structuring of tasks is a purely static matter and current operating systems do not allow dynamic reconfiguration in the task structure, the necessary activities have to be carried out at design time and cannot be automated. The restructuring of tasks is another example to improve the system dependability at expense of resource optimization.

So far in the space partitioning, only the direct data access by means of read/write operation is discussed. As depicted in Figure 9-19, the indirect memory access to variable can happen with function call of a runnable in task 2 in the 2nd situation. Because this runnable is originally allowed to change the variable, no error will be reported. The allocation of runnables/functions and the access to a certain driver interface should be also partitioned and protected as to the memory as well. In order to prevent the scenario in Figure 9-19, the runnable will temporally lose the unlimited authority as trusted object to the variable by inherit the status of “untrusted” from task1 when it is called by task1.

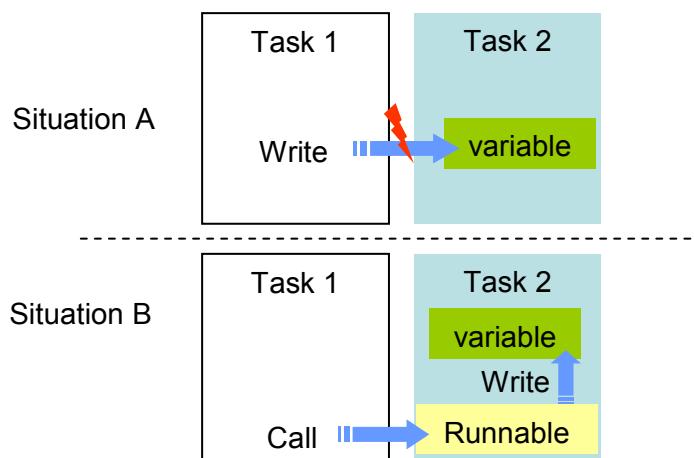


Figure 9-19: Memory protection of indirect data access

9.3.2.3 Time partitioning with OS and Software Watchdog service

To face the challenges stated in Subsection 9.3.2.1, time partitioning to monitor individual application software components is required to improve the overall system dependability [Tin03]. Time partitioning of application SW-Cs implies again an integrated approach by monitoring runtime behavior from real time and logical control flow requirements. In the following subsections, the designs for the time portioning will be discussed around these two aspects.

9.3.2.3.1 Run-time monitoring with Operating System

From the point of view in the software architecture, operating system is the best suitable basic software module to monitor the timing behavior of running entities.

- In the OSEK-compatible operating systems, as mentioned above, all the OS-objects are static configured and structured, in another word, they are all known to the operating system.
- Scheduling service is one of the basic services in the OSEK-compatible operating systems. Time services for the scheduling can be directly applied for the time monitoring.
- As discussed in Subsection 9.3.2.2, for the space portioning service all the untrusted OS-objects are associated with a look-up table in the OS for the space partition services. If this table can be extended with the timing parameter, they can be monitored by the OS as well.

In the following, a few terms used to discuss the runtime monitoring in operating system will be briefly explained:

Execution Time	As depicted in Figure 9-20, the execution time refers to the time, an OS-object spends in the RUNNING state (task state as defined in OSEK OS 2.1 [OSK05]) without entering the SUSPENDED or WAITING state. For ISRs it is the time from the first to the last instruction. For Tasks/ISRs it excludes all preemptions due to higher priority tasks/ISRs executing in preference. The execution time includes the time spent in the error, pretask and posttask hooks and the time spent making OS service calls.
----------------	---

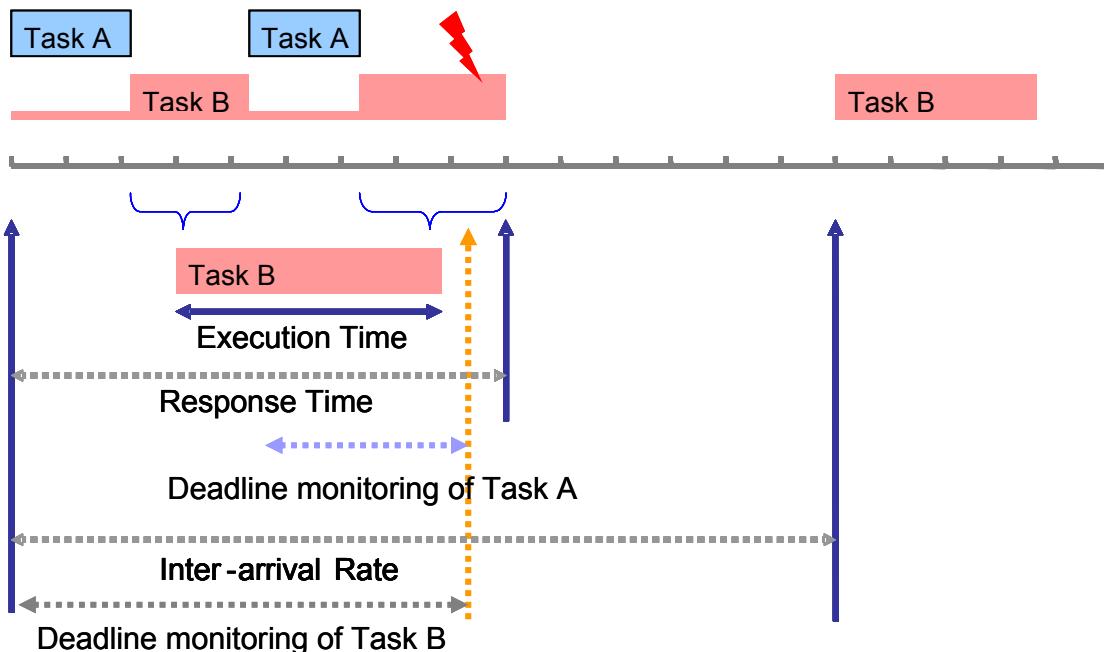


Figure 9-20: Definition of task execution time

Execution Time Budget	Maximum permitted execution time of an OS OS-object. Typically, this will be the worst case execution time (WCET) of a Task/ISR. For tasks the WCET budget counter is suspended if the task enters the WAITING or SUSPENDED state, and continued after the task is in READY state again.
Protection Error	Systematic error in the software of an OS-Application. <ul style="list-style-type: none"> ▪ Memory access violation: A protection error caused by access to an address in a manner for which no access right exists. ▪ Timing fault: A protection error that violates the timing protection.

In Figure 9-20, the constraints of the traditional approach of deadline monitoring in OSEKtime ([OSK01] and [HRK00]) is shown. The task B could violate the deadline, because task A, was e.g. executed too often and preempts the task B. In such a situation, the deadline monitoring based on WCET of task A will not report any error, while the task B, which is running properly, will be identified as faulty by the deadline monitoring. To solve such a problem, it is more reasonable to monitor the task execution time according to the preconfigured execution time budget as defined above to discover the real faulty task. Similar as designed in the OSEKtime, an OS interrupt can be triggered to report the timing fault for further fault treatment.

9.3.2.3.2 Run-time monitoring with Software Watchdog

Again as depicted in Figure 9-20, with execution time monitoring, faulty tasks, which take too much system resource of computation time can be detected. But as demonstrated in the example, if the task A has a priority high enough, it can preempt/block other tasks with the same or lower priority. Each time when the task A is executed, it fulfils its own requirement of task execution time, while keeping other tasks fail to reach the deadline. The faulty task A, even if it is executed much more often than normal, could stay undetected with task execution time monitoring. In the worst case, task B with lower priority can theoretically never report error with execution time monitoring but never finishes its job.

As discussed in Figure 9-17 of Subsection 9.3.2.2, tasks can be made up of runnables from different applications. So far the granularity of fault detection on the layer of task is the deadline monitoring and execution time monitoring, which are not fine enough for fault detection of runnables. As two sides of a coin, we could use a dedicated dependability software service to monitor the execution frequency of a runnable with aliveness indication to guarantee that the runnable is still running but not executed too often. In order to reduce the complexity and save the computation overhead, not all the runnables should be monitored, we can choose the runnables from the critical program flow in the following aspects

- Input signals collection
- Preparation of input data signals
- Signal processing and evaluation/plausibilisation of the results
- Control of the actuators

as the monitored objects. As mentioned above, usually these runnables are also mapped to tasks with higher priority.

Time partitioning also includes the correct logical control flow of applications, which is highly recommended in [ISO26] Table 6.5 for ASIL-D. As shown in Figure 9-21, in the faulty situation runnable R2 is branched. In case of a permanent branch fault, the aliveness fault of R2 can be detected by the aliveness monitoring of the runnable but it can be detected sooner with a runnable control flow monitoring. The control flow error might be detected with the execution time monitoring or deadline monitoring, but only to a much later time point.

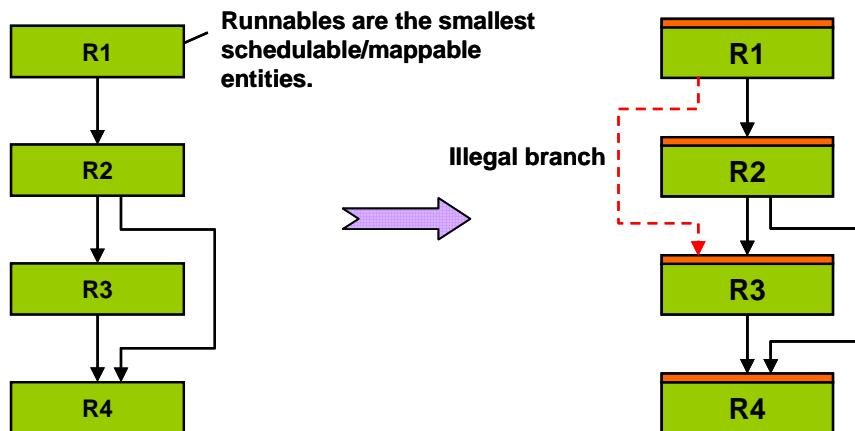


Figure 9-21: Control flow check of runnables

In the field of control flow checking, a lot of research has been carried out in the IT-industry to check the correctness of program flow. Most of the current methods of control flow checking are based on assigning signatures to blocks as introduced in [Nah02]. Such a technique suffers from high performance overhead and low flexibility with regard to modification of programs.

Software Watchdog here is supposed to ensure the real-time characteristics of the system as an extension to the hardware watchdog mentioned in Chapter 8 to face the challenges of increasing density of application SW-Cs. The following functionalities are planned to be implemented within the framework of the SW-Watchdog service:

- Monitoring the heartbeat of individual runnables including aliveness and arrival rate monitoring.
- Detecting control flow errors: control flow errors are unexpected changes in the sequence of instructions executed by the main processor. Taking the complexity into consideration, only the execution sequences of runnables are supervised by the SW-Watchdog.
- Gathering the detected error information from the two above mentioned services, compare the number of the detected errors to the threshold and derive error indication states of the tasks and send them to fault treatment.

The interactions between the SW-Watchdog service and its runtime environment are specified as following. The dependability software architecture specified in Subsection 9.2 represents the environment, where the SW-Watchdog service is embedded and integrated. The major information, such as heartbeats and execution information of runnables will be sent to the SW-

Watchdog through Common/Domain-specific and ISS interface. And the error report from SW-Watchdog to Fault Management Framework (FMF) will be specified by FMF standard interface

In the following the functional structure of SW-Watchdog is specified in three basic blocks:

- Heartbeat monitoring: With the help of the heartbeat indication routine, runnables send their heartbeats to the SW-Watchdog. SW-watchdog supervises those heartbeats and checks if runnables are executed frequently enough (aliveness monitoring) or if they are executed too frequently (arrival rate monitoring).
- Control flow checking (CFC): The control flow checking service monitors the execution sequence of runnables by comparing real executed successors with the possible successor set of the predecessors.
- Task State Indication Unit (TSIU): To provide a proper tolerance of the errors detected in the upper two blocks, they should be sent to the TSIU. And TSIU should compare the number of the detected errors to the threshold and derivate error indication states of the tasks and send them to FMF.

The Figure 9-22 shows the functional structure of the SW-Watchdog.

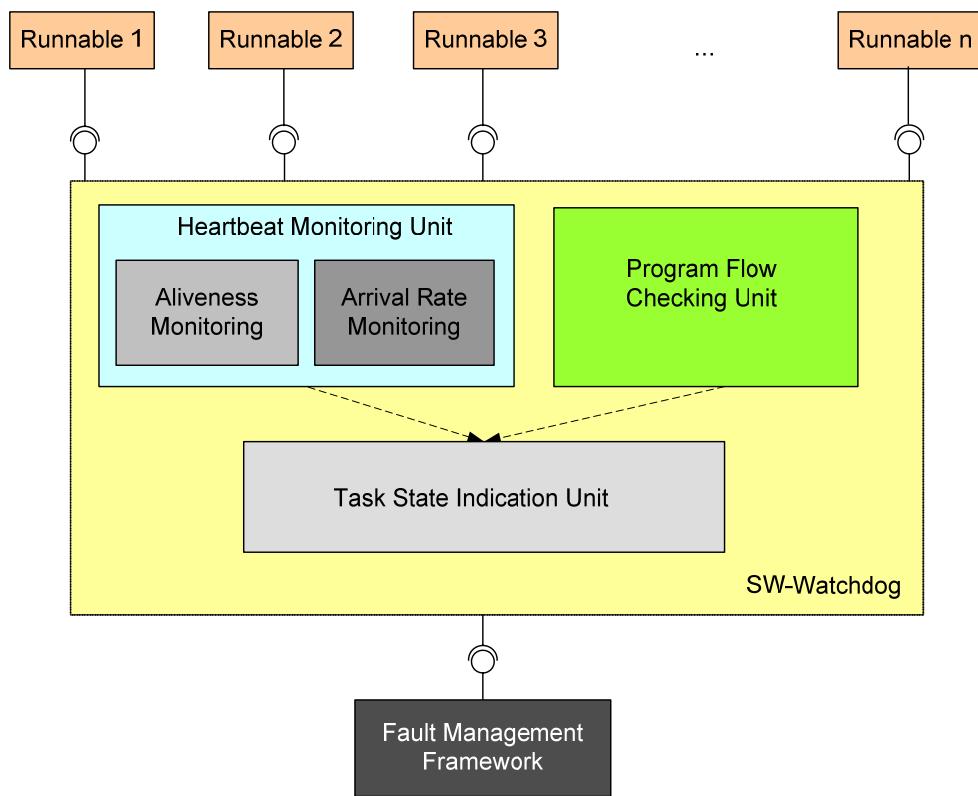


Figure 9-22: Different units constituting the SW-Watchdog service

In the following sections the functionalities of SW-Watchdog will be introduced in more details.

Heartbeat Monitoring

In order to setup Heartbeat Monitoring the attributes of each monitored runnable must be provided at the configuration phase. The attributes in this context means parameters, which describes the boundary of the correct execution. There are five parameters required, as listed in Table 9-16.

Parameters	Description
m	Minimal number of expected heartbeats within n cycles
n	Predefined number of monitoring cycles in the fault assumption for aliveness monitoring
p	Maximal number of expected heartbeats within q cycles
q	Predefined number of monitoring cycles in the fault assumption for arrival rate monitoring
<i>runnableID</i>	ID of runnable.

Table 9-16: Parameters for runnable in Heartbeat Monitoring

After specifying the parameters of a runnable, on the side of SW-Watchdog some data resource for the monitoring are required too, as listed in Table 9-17.

Data resource	Description
AC	Aliveness Counter, track the aliveness of monitored runnable
CCA	Cycle Counter for Aliveness, record the monitored cycles of aliveness
ARC	Arrival Rate Counter, track the arrival rate of monitored runnable
CCAR	Cycle Counter for Arrival Rate, record the monitored cycles of arrival rate

Table 9-17: Data resource for runnable in Heartbeat Monitoring

The interactions between the current runnable and heartbeat monitoring are implemented through heartbeats. Heartbeat will be sent, when the runnable is executed. After the heartbeat arrives in heartbeat monitoring, the data resource, such as AC and ARC, will be updated or in other words incremented by one.

On the other side Heartbeat Monitoring will update their own counters (e.g. CCA and CCAR) at the beginning of each monitoring cycle. It must be ensured that the monitoring cycle is independent from the cycle, of which aliveness is sent, and within one monitoring cycle all monitored runnables will be checked against their predefined boundary. According to the fault assumption two services must be implemented: Aliveness Monitoring and Arrival Rate Monitoring. Aliveness Monitoring

checks, whether the runnable is out of its lower limit m , at the same time Arrival Rate Monitoring checks the upper limit p of runnable. Once the faulty runnable is detected, the ID of the faulty runnable will be reported to TSIU.

The last step of the monitoring is to decide, whether the counters (e.g. AC and CCA, ARC and CCAR) are needed to be reset. At the Aliveness Monitoring AC and CCA will be reset, when the predefined number of monitoring cycles has expired. Arrival Rate Monitoring is slightly different, the ARC and CCAR should be reset at the end of each cycle, if a fault of the arrival rate monitoring is determined at the end of this cycle. The following figures show how Aliveness Monitoring and Arrival Rate Monitoring work.

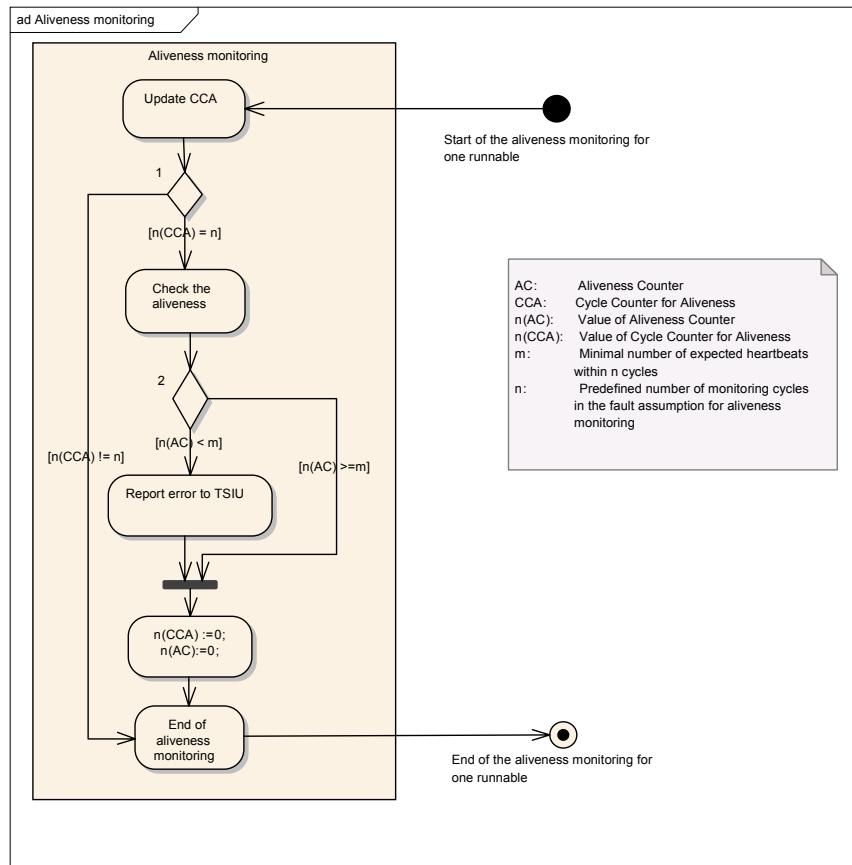


Figure 9-23: Activity diagram for Aliveness Monitoring

As demonstrated in Figure 9-23, after starting aliveness monitoring for one runnable, the CCA will be incremented by one, then we will compare CCA with predefined number of monitoring cycle n (see condition 1 depicted in Figure 9-23 as $\diamond 1$). If CCA is not equal n , aliveness monitoring for one runnable will be ended. In the case that CCA and n are equal, we will check the aliveness counter AC of the runnable (see condition 2 in Figure 9-23), whether it reached the minimal number of expected heartbeat m . If the expected heartbeat is not reached, an aliveness monitoring error will be reported to FMF. Before end of aliveness monitoring CCA and AC will be reset to zero.

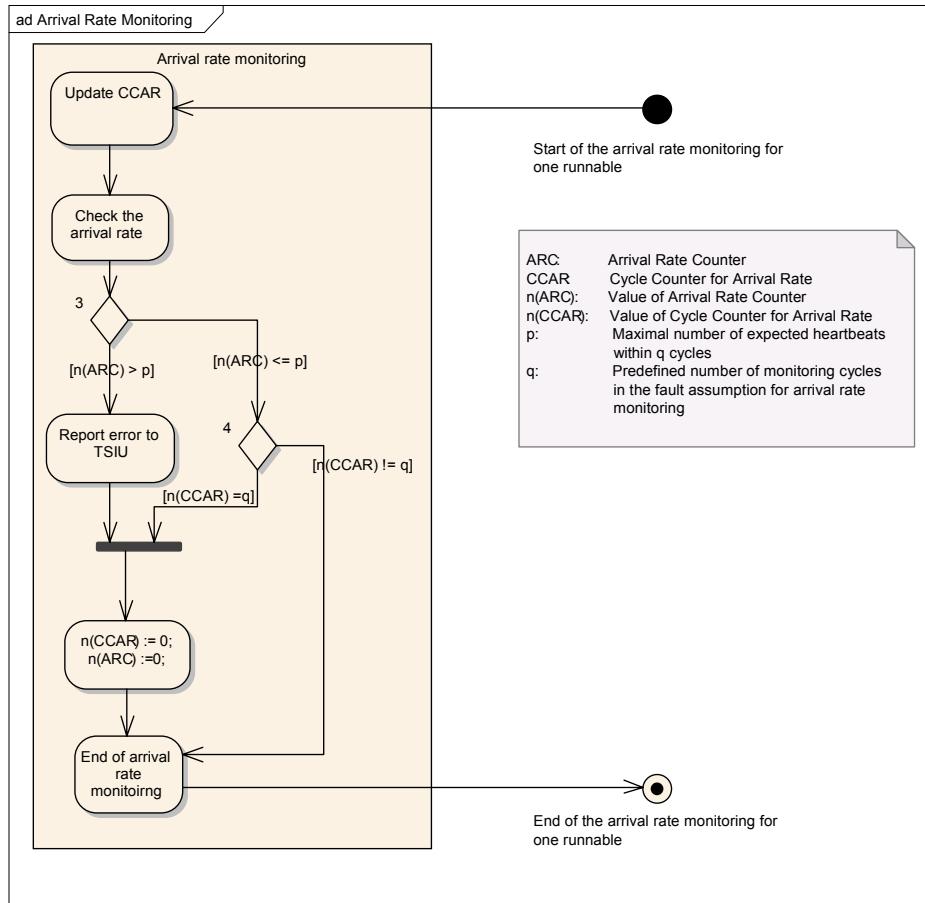


Figure 9-24: Activity diagram for Arrival Rate Monitoring

The design of Arrival Rate Monitoring is slightly different compared to the design of Aliveness Monitoring, as depicted in Figure 9-24. The first step of Arrival Rate Monitoring is also to update the cycle counter CCAR, but CCAR will not directly compare with predefined number of monitoring cycle q . We will check first the arrival rate counter ARC , whether it is greater than the maximal expected heartbeats p . In the case that ARC is greater than p , an error of arrival rate monitoring will be reported to TSIU. In other case if ARC equals or is smaller than p , another comparison, CCAR with predefined monitoring cycle q , will be carried out. If CCAR is equal to q , CCAR and ARC will be reset to zero. Otherwise arrival rate monitoring will be ended.

To demonstrate the concept and algorithms of Heartbeat Monitoring, an example will be given here. For a better overview, this example only demonstrates the supervision of one runnable.

For this runnable the fault assumption is:

- There are less than 2 heartbeats from the monitored runnable in 3 monitoring cycles, $m=2$, $n=3$.
- There are more than 3 heartbeats from the monitored runnable within 2 monitoring cycles, $p=3$, $q=2$.

Therefore, the examination terms of the heartbeat monitoring are:

$$n(AC) \geq 2 \text{ if } n(CCA) = 3$$

$$n(ARC) \leq 3 \text{ for all } n(CCAR) = 1 \dots 2$$

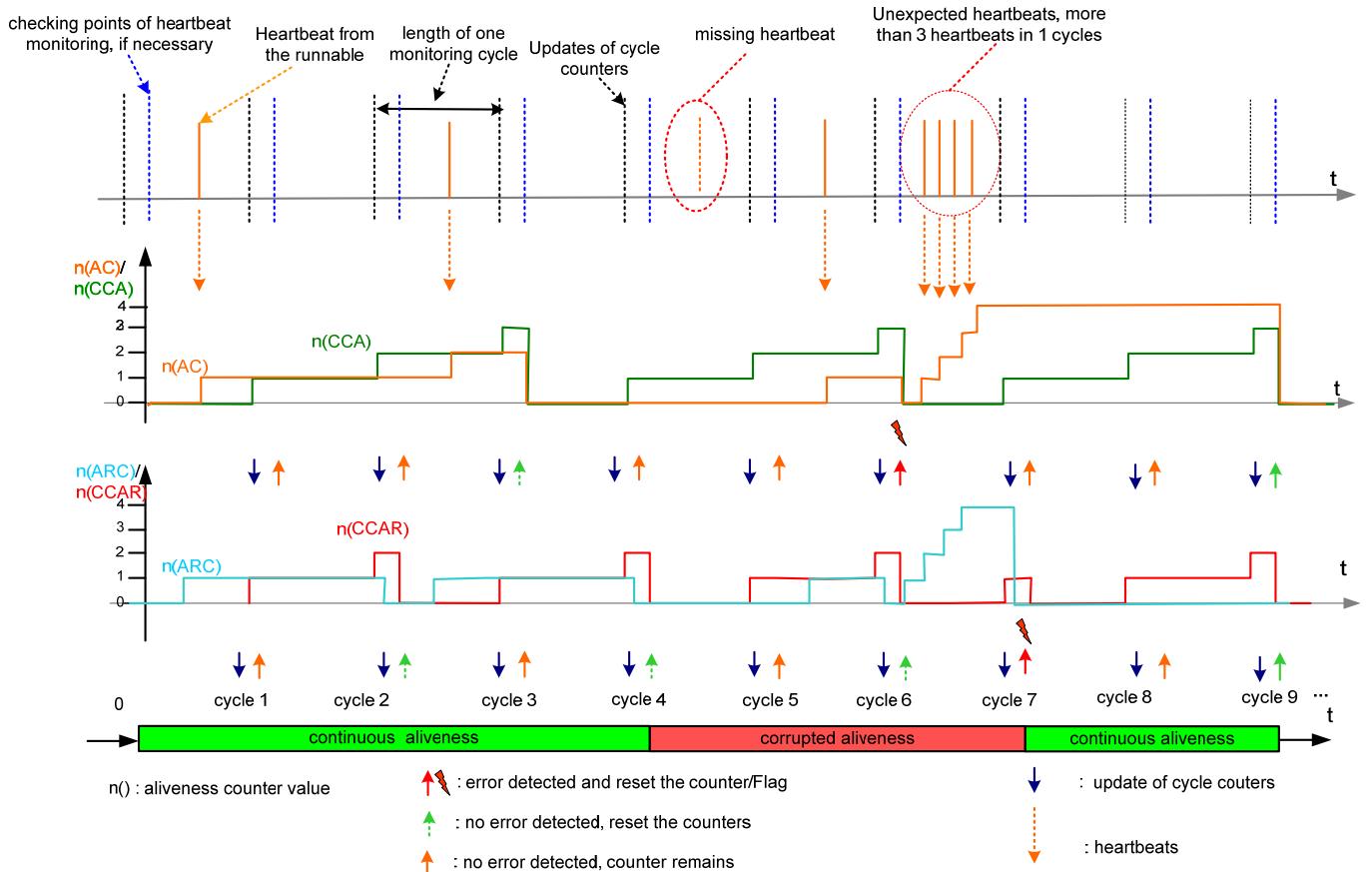


Figure 9-25: Example of Heartbeat Monitoring

In the simple example of heartbeat monitoring in Figure 9-25, it is assumed that all the counters have the initial value 0 at the beginning of the selected monitoring period. The work of SW-Watchdog in the following 9 cycles, is explained in detail:

- Cycle 1 to cycle 4, and cycle 8 to 9: There are no missing and unexpected heartbeats.
→ Checking criterion fulfilled. AC and CCA are reset at each 3 cycles, while ARC and CCAR are reset at each 2 cycles.
- Cycle 5: there is a missing heartbeat in this cycle, since the value of CCA has not yet reached 3.
→ No fault is detected and AC retains its current value.
- Cycle 6: There is only one heartbeat in cycle 4, 5 and 6. And the value of CCA reaches 3 at the end of cycle 6.
→ A fault in the aliveness monitoring is reported, because CCA reaches 3 while AC=1, CCA and AC are reset at the end of cycle 6,

- Cycle 2, 4 and 6: When CCAR reaches 2 (defined as q=2)
 - CCAR and ARC are reset. The ARC are checked at each cycles (yellow arrows), while the reset is only executed at each 2 cycles if no fault is found.
- Cycle 7: More than 3 aliveness indications are sent by the runnable.
 - ARC is over 3 and a fault is detected. ARC and CCAR are reset to their initial value 0.

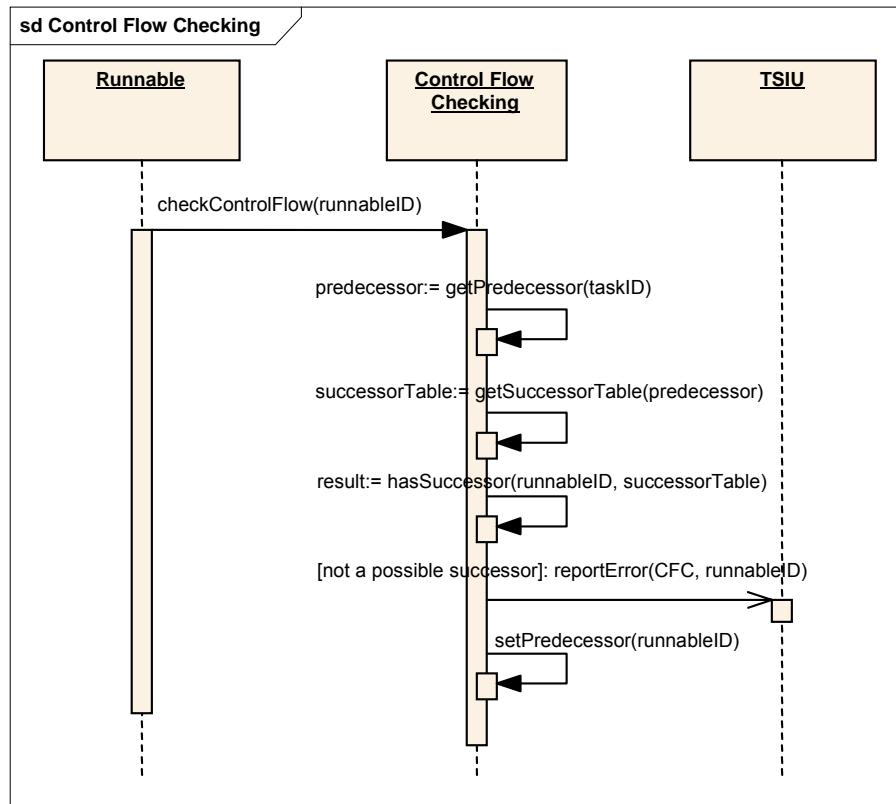
Control Flow Checking

Another service provided by SW-Watchdog is Control Flow Checking. In comparison to traditional definition of Control Flow Checking in IT, which controls the correct execution of computer programs, here we only focus on the correct execution sequence of runnables in critical tasks. Before describing the internal mechanism of Control Flow Checking, some information about runnables contained within the critical task, must be prepared at configuration phase, as listed in Table 9-18.

Parameters	Description
<i>successorTable</i>	Table of runnable ID, which specify all possible successors belonged to the critical runnable.

Table 9-18: Parameters for runnable in Control Flow Checking

For each safety-critical task, Control Flow Checking will provide a predecessor indicator to record the last executed runnable of this task, so that the type of predecessor indicator must be the same type as runnable ID. Once a critical runnable moves into the running state, it will notify SW-Watchdog by using its service *checkControlFlow*. After that Control Flow Checking will examine whether the current runnable belongs to the possible successor set of last executed runnable. If the current runnable is not a possible successor, an error will be reported to Task State Indication Unit by using the service *reportError*, as depicted in Figure 9-26. After checking the control flow the predecessor indicator will be updated with the current runnable.

**Figure 9-26: Sequence diagram for Control Flow Checking**

Task State Indication Unit

The Task State Indication Unit must provide two vectors for each critical task, a condition vector and an error indication vector. The condition vector contains predefined bounds for each sort of error and it should be specified at the configuration phase. The error indication vectors are designed to record the results reported by heartbeat monitoring and control flow checking of a certain task. Each error indication vector contains three elements. The first element is the number of detected aliveness errors. It is the sum of the aliveness errors detected by the aliveness monitoring. Likewise, the second and third element should be the number of arrival rate errors and control flow errors. The determination of error indication state of a task is based on comparisons of condition vector with error indication vector. The two vectors are depicted in Table 9-19.

Name	1 st element	2 nd Element	3 rd Element
conditionVector	Max. errors of Aliveness Monitoring	Max. errors of Arrival Rate Monitoring	Max. errors of Control Flow Checking
errorIndicationVector	Errors of Aliveness Monitoring	Errors of Arrival Rate Monitoring	Errors of Control Flow Checking

Table 9-19: Two vectors of TSIU

Once an error of one runnable in a task is reported, one element of error indication vector should be incremented depended on the error type, then the error indication vector will be compared with

the condition vector of the same task. As long as one element of indication vector is greater than the one in condition vector, the error indication state of the task will be set to “faulty”, and the faulty state will be sent to FMF by using service *indicateTaskState*. Since the state “faulty” is already sent to the FMF, the error indication vector must be reset to zero, in order to record new error for the following supervision, as depicted in Figure 9-27.

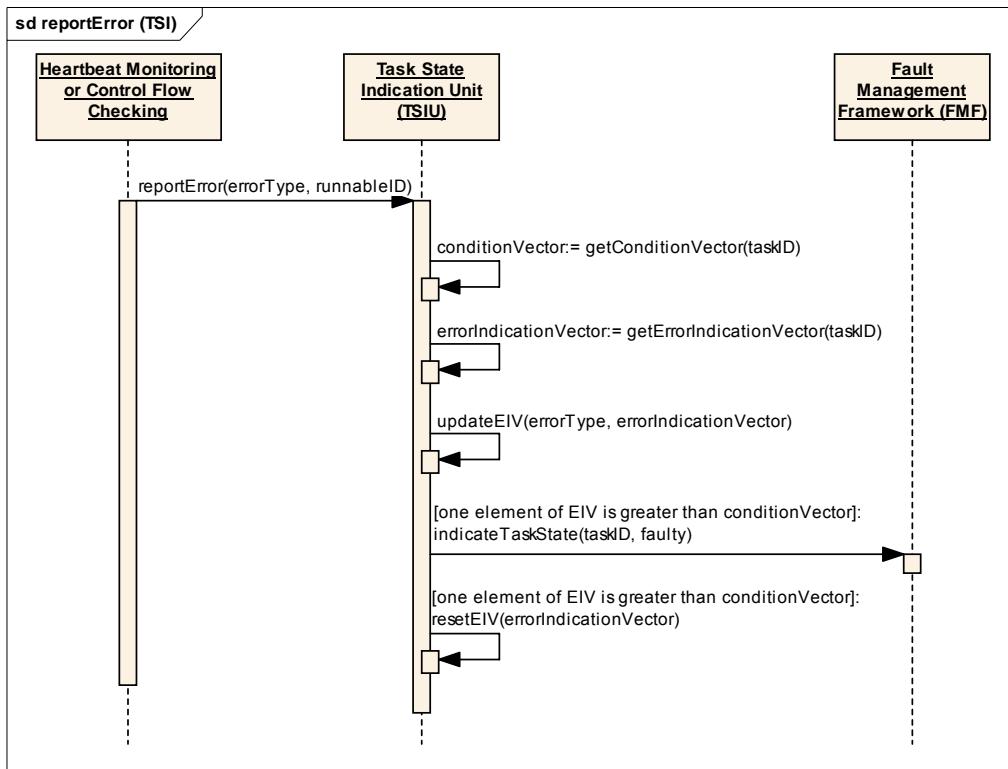


Figure 9-27: Sequence diagram for TSIU

Configuration of SW-Watchdog

In order to setup the SW-Watchdog a few parameters of SW-Watchdog must be specified at the configuration phase. Based on their application scopes, they can be split mainly into three categories.

- The first category contains the information about configuration of the Heartbeat Monitoring for each monitored runnable. The parameters m and p give the low limit and upper limit of heartbeats within predefined monitoring cycles (e.g. n and q), and this information should be provided by application designer. Another basic information is *runnableID*, which is unique in the whole system and consists of ID of the task, ID of the application and ID of runnable it self. Thus the *runnableID* will not be given by the application designer; it should be generated automatically by design tools.
- The second category contains the information about configuration of the Control Flow Checking. For Control Flow Checking each critical runnable should have a *successorTable*, which describes all the possible successors of the critical runnable. Since a runnable does not know its successors, until the mapping process of runnable on the task is finished, this information should also be generated automatically and provided by design tools.

- The last category is configuration of Task State Indication Unit (TSIU). For each task the *conditionVector* must be defined, which gives the bound of maximal tolerant faults. This bound is mostly depending on specific application design and system requirement, thus this parameter must be specified by the application designer.

As a summary to the Subsection 9.3.2.3, time partitioning services as OS with MPU, Protected Application service and SW-Watchdog service will improve on one side the system dependability with runtime monitoring. On the other side, from the view of a system integrator, they also contribute to locate and analyze the error at the system integration of functions from different suppliers, e.g. they can help OEM to determine the supplier that is actually responsible for the malfunction.

9.3.3 Dependability services of fault treatment

In the last subsections we discussed quite a few dependability software services for the individual monitoring and fault detection of application SW-Cs. Again with increasing intensity of application SW-Cs, it is required to gather these fault detection information, to management and filter out the safety relevant information. Thus an application wide system state can be pictured and an application wide coordinated fault treatment can be carried out.

9.3.3.1 Fault Management Framework

Functional design

The FMF ([FMF06] and [EiK07]) was designed for fault detection and subsequent compensation or recovery. This later aspect includes fault management (identify fault scenario) and fault treatment (identify and conduct fault mitigation strategy). Further on, a logging service was integrated into the FMF. All these aspects are described briefly in the following:

- Fault detection: Generally speaking, fault detection is often highly application specific, however, generic fault detection services (e.g. Software Watchdog) can be provided by the ISS framework. Here the task of the FMF is to handle faults and errors reported by the Software Watchdog.
- Fault management: Similar to fault detection, fault management is often application specific as well, but the FMF can offer some generic fault management mechanisms so that the handling of faults can be separated from the nominal task and the application programmer can concentrate on the application. Hereby the FMF provides general services allowing for applications (as well as the dependable software platform in Subsection 9.2) to obtain a consistent and global view of the fault state of the ECU as well as of individual applications on the ECU. This information can be used for isolation and diagnosis purposes as well as to make decisions on appropriate recovery strategies.
- Fault treatment: Based on predefined patterns from the application developers and system integrators, the FMF can combine these local (or even distributed) fault detections to derive a global view of the fault state of the ECU and the applications. Based on this global view, recovery actions could be initiated according to predefined principles. The fault-treatment measures are statically defined by the application developers and system integrators, since

a deterministic fault tolerant behavior is desired for the safety applications. Such a design brings also the constraints that only faults, which are considered in the fault hypothesis in the design phase, can be treated during runtime, since the fault tolerant system can only tolerate the faults which were considered in the design. Transient faults (e.g. transient faults from EMV) should be filtered by the FMF when the application functionalities are not affected.

- Logging service: It is also important to store appropriate information (“Logging”) concerning the various faults and actions for later use in, e.g., workshops, so as to enable off-line analysis leading to fault assessment and repair, and maybe even future prevention.

In order to achieve these functionalities as discussed above, the FMF was structured into four functional units as can be seen from Figure 9-28 [FMF06].

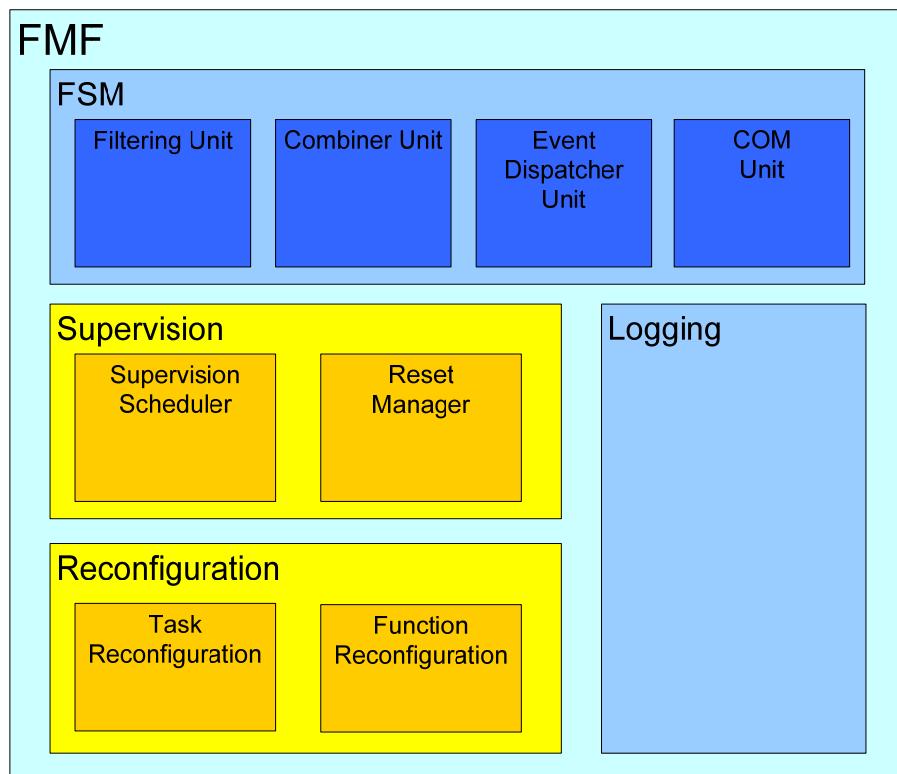


Figure 9-28: Fault Management Framework Structure

9.3.3.1.1 *Fault State Manager*

FSM is the core of the FMF since it receives the error reported from the software component and dispatches events to other units and software components. The FSM also manages the state of each error: present, confirmed, memorized and any combination of the three. The FSM performs error filtering, detection of error combinations, error event dispatching and distribution of error information through its four sub units.

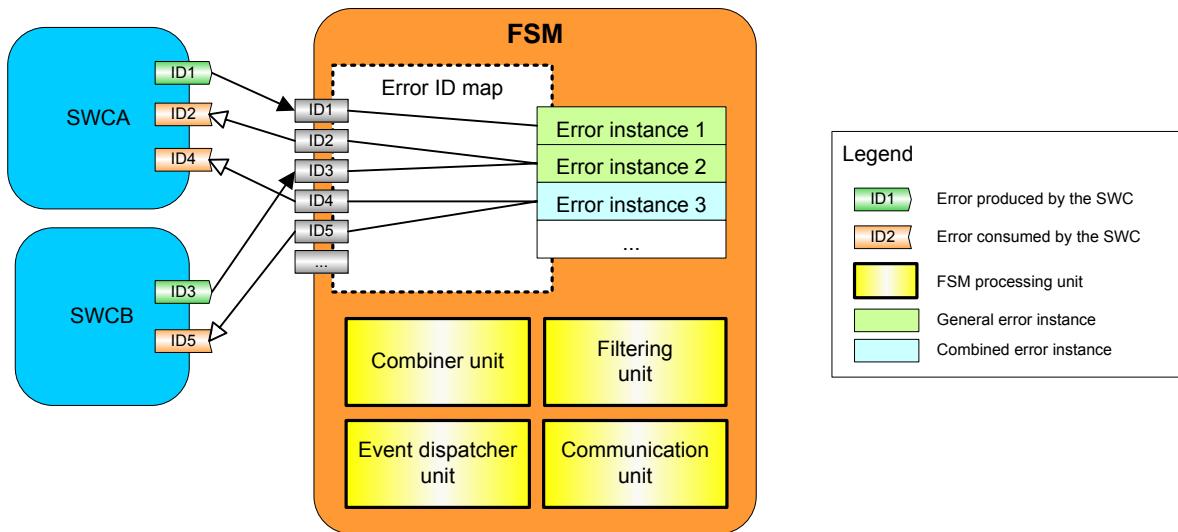


Figure 9-29: FSM structure and ID mapping

As depicted in Figure 9-29, each SW-C could have two kinds of interactions with the FSM:

- They can report error status to the FSM, those errors are called produced errors
- They can read the status of error from the FSM, those errors are called consumed errors

The mapping between the errors IDs used by the SW-Cs and the error instances is defined in the FSM configuration. This mapping allows reducing the coupling between SW-Cs, therefore enables a more flexible software architecture.

1. The filtering unit handles the filtering algorithms (e.g. counter filter or sliding window based a stability filter)
2. The combiner unit is responsible for managing combined error instances. When an error is reported by a SW-C, the corresponding mechanism for combined errors is the evaluation of its logical expression. The evaluation of combined errors must respect the following rules:
 - A combined error has to be evaluated each time one of the error instances or its reference is updated (e.g. consecutively to a report or time based filtering), and during the same processing session.
 - A combined error is evaluated only once per execution cycle.

The evaluation of a combined error instance also includes its update, so the first rule implies that each time an error instance is updated, all combined errors, which depend on it directly or indirectly, have to be evaluated during the same execution cycle. This means that any update can trigger a chain of evaluation of combiner errors. The combiner handles those chained evaluations as following:

- Each time when an error instance is updated outside the combiner, the combiner is notified of the update and marks all the dependent combined errors as pending.
- At the end of an execution cycle, just before running the event manager, the combiner is called / run to evaluate all the pending combined errors, clearing the pending flag at each evaluation.

3. The event dispatcher in FSM as shown in Figure 9-30 is responsible for dispatching error events to SW-Cs and other FMF services. A given error event is linked to its consumers through event handlers, which define the notification mechanisms to be used and its parameters. There are three main types of event handler:

- Synchronous event handlers which treat the event in the same execution cycle the event is detected. One example mechanism is a callback function which is called each time the event is dispatched.
- Asynchronous event handlers which are notified of the event but do not treat it in an error processing session. One example mechanism is to increase a counter to signal that the event has occurred.
- Supervision event handler: this is a special synchronous event handler dedicated for supervision, there can only be one such handler in the FSM event dispatcher. Its role is to run the Supervision's own scheduler which dispatches itself the events to its Fault Treatment Units.

Similar as the combiner unit, the event dispatcher uses a delayed execution/dispatching pattern:

- Each time the confirmed status of an error instance changes, the dispatcher is notified of the event and it marks the event and its associated handlers as pending.
- At the end of a processing session the dispatcher will be executed, e.g. it notifies all pending handlers and after that clears all the pending flags. In this way, event handlers which are activated during a given session can be triggered in the correct order.

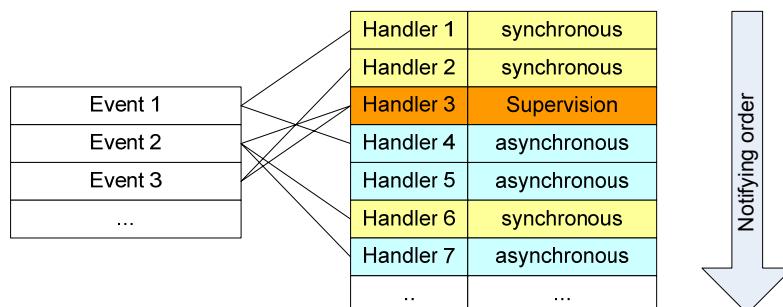


Figure 9-30: Event dispatcher example

4. The communication unit is responsible for distributing error information between the ECU local FMFs for a system wide distributed fault treatment.

One of the aims of FMF is to manage the application wide coordinated fault treatment. Since the application SW-C can be mapped to different ECUs, the FMF should be distributed as well with each ECU runs its own local copy FMF. This means that there is no central fault management ECU. The FMF provides a mechanism to distribute error information. The FMF also allows the deployment of local and system fault treatment strategies through its Supervision and Reconfiguration units.

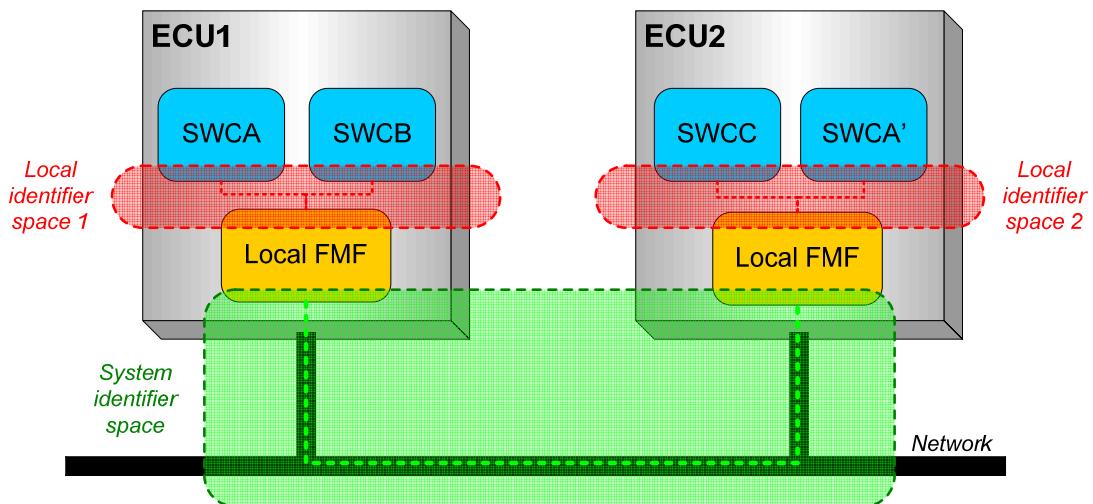


Figure 9-31: HW/SW mapping of the FMF

As depicted in Figure 9-31, there are two levels of communication in the FMF:

- Local level: communications between the SW-C and the local FMF
- System level: communications between different local FMFs

The two levels are kept separated, this means in particular that error information is never sent by a SW-C to a remote FMF, and it should always go through the local FMF instead.

For its inter ECU communication, the FMF relies on the COM-services. The communication requirements are divided in two categories:

- The requirements for error information distribution: the bandwidth requirements mostly depend on the number of system level errors, the timing requirement for each error and communication protocol used. For example the use of Agreement Protocol may require a higher bandwidth.
- The requirements for system fault treatment strategies: the strategies are completely up to the application / system designer and so are their communication requirements.

In order to avoid a modification on one ECU to impact the FMF instances on all the other ECUs, error identification is always local to an identifier space. Each local FMF has its own local identifier space to communicate with local SW-C, and there is one system identification space as shown in Figure 9-31. The identifier spaces are kept separated, so that the error identifiers of one space are not visible from the other space. The mapping between the identifier spaces is defined by the local FMF configuration. A local FMF only has the visibility on its local identifier space and the system identifier space, therefore it is not possible to map directly two local identifier spaces, and they must be linked through the system identifier space.

To clearly differentiate the local identifier spaces from the system identifier space, the FMF uses different error ID types:

- System level error identification: it is based on global identifiers which are used to identify an error uniquely within the system. Contrary to the local IDs, global IDs are not visible from the software component as they are defined internally in the FMF configuration.

- ECU level error identification: local identifiers are the only ones to be directly visible by the software components. Those local identifiers enable the local FMF to identify errors uniquely within the ECU.

9.3.3.1.2 Supervision Unit

The role of Supervision Unit is to supervise the execution of the fault treatment units. An FTU (Fault Treatment Unit as depicted in Figure 9-32) can be a sub module of a SW-C, or a completely standalone service, but it is managed / executed only by the Supervision Unit, and for that purpose it must have a special interface. Contrary to other SW-Cs, the FTUs can access the FMF reconfiguration services.

The entity responsible for managing the FTUs is the supervision scheduler. It processes error events (from the FSM), system events (startup, timer, etc.) and dispatches them to the FTUs.

The last important entity of supervision is the reset manager. The reset manager allows the FTUs to make a reset request or inhibit reset requests.

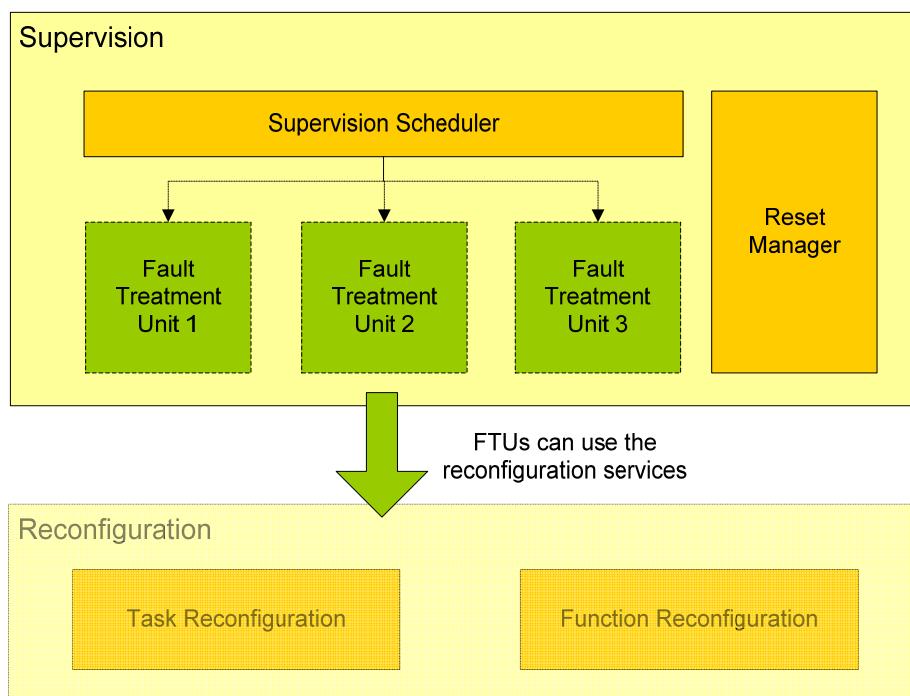


Figure 9-32: Overview of supervision

Supervision scheduler

The supervision scheduler has a similar role as the OS scheduler. The main difference is that where the OS manages the execution of tasks, the supervision scheduler manages the execution of the FTUs event handlers. Also the supervision scheduler forbids reentrancy. This centralization of the execution of the FTUs allows having a better control over the coherency and determinism of the integration of the different fault treatment strategies.

Fault Treatment Unit

The FTUs implement the fault treatment strategies. An FTU is actually implemented by the application programmer but it has specific characteristics which differentiate it from an App.SW-C. First an FTU can access the reconfiguration services and the reset manager. Secondly an FTU can only be executed through its event handlers and by the supervision scheduler. Even though the content of an FTU is application dependant, FTUs can be divided in two categories:

- Local FTUs which only interact with SW-Cs and FTUs located on the same ECU.
- System FTUs which also interact with system FTUs located on other ECUs.

9.3.3.1.3 Reconfiguration Unit

The role of the Reconfiguration Unit is to provide mechanisms to reconfigure functions, runnables and tasks of the ECU. Functions can be activated or deactivated while tasks can additionally be restarted. The services provided by this unit can only be used directly by the Supervision Unit. More details about reconfiguration can be found in Subsection 9.3.3.2.

9.3.3.1.4 Logging Unit

The logging unit is responsible for storing errors and freezing associated frame data in the non volatile memory, which is especially useful for later fault removal and system recovery after the ECU restart. The logging unit interacts with the following software services: the non volatile memory manager, the diagnosis and calibration service.

9.3.3.2 Fault treatment with dynamic configuration

Compared with the IT-Industry, where the dynamic reconfiguration is already a common practice for the system recovery and update, state of the art in the embedded in-vehicle electronics is still a statically configured system. There are various reasons for that [DyR96]:

- As its name implies, embedded electronics are embedded together with the hardware. Once a system is embedded, there might be no reason to change or update it.
- In the embedded system, resources such as processor power and memory are relative limited.
- Safety relevant systems usually require hard real-time requirement. Reconfiguration, instead, can require more time than allowed.
- A deterministic behavior rather than a dynamic configuration is vital for the safety relevant applications.

As have mentioned in Subsection 3.4.4, the same evolution in the IT-Industry will take place in the automotive embedded electronic, where the reconfiguration will be a more and more important topic. The motivation and scope of the dynamic reconfiguration in automotive electronics, however, are quite different to the IT-industry:

- Enabling technologies such as common software architecture, model-based software design and component-based software services (esp. support from RTOS) are becoming mature technologies.
- Product platform strategy: It is possible for an entire product line to use the same software core, perhaps stored as a library in an EPROM. Based on the hardware configuration and desired functionality, the software can easily be band-end configured for each individual product model. This eliminates the need to develop and maintain separate software for each different product, esp. for the automotive supplier, dynamic reconfiguration is becoming more and more interesting to support different vehicle models on the same platform of one OEM or even for different OEMs.
- More availability with dynamic reconfiguration and fail-degradation: State of the art in the in-vehicle safety system is fail-safe and fail-silent. As introduced in Subsection 2.1, dependability is a balance of different aspects. In order to improve the availability while keeping the system safe as a trade-off, dynamic reconfiguration with fail-degradation is a proven approach, esp. for semi-X-by-Wire application.
- Remote upgrades and maintenance: Compared with the life cycle of 4 to 8 years of traditional pure mechanical vehicles, the life cycle of electronics of 1-3 years is much shorter [McK05]. “Patches” can be downloaded / updated to the customer to fix the bugs and introduce more features/extensions.

It is clear that only systems which consist of several components can be reconfigured; these components can be hardware or software. Also it is clear that reconfiguration requires some kind of redundancy, which is not an identical ECU in every case, but it can be any kind of hardware or software unit that can, at least partly, take over the duty of the faulty unit.

A reconfiguration in software usually changes the flow of information between the components and/or changes the purpose of components. With these actions the functional mapping is redefined for some or all components. The aim of the reconfiguration is a system that consists only of error-free components and that exhibits the maximum of functionality possible (fail-degradation).

Reconfiguration can be static or dynamic. Static reconfiguration appears in the development phase of a system, dynamic reconfiguration occurs during runtime. The reconfiguration mentioned here refers mainly to the dynamic reconfiguration in software, which contributes to the fault tolerance, but does not address fault repairing. Dynamic reconfiguration defined in [IEC07] C3.13 “remapping the logical architecture back onto the restricted resources left functioning” with “run-time redundancy” is not recommended for ASIL-C and ASL-D applications according to [IEC03] Table A.2 about software architecture design, thus dynamic reconfiguration with resource reallocation is not in the scope of work here.

Two types of events can trigger a dynamic reconfiguration for the fault tolerance:

1. The occurrence of a fault causing an error
2. The disappearance of an error in a component.

In the first case three actions have to take place:

- The fault has to be detected and confirmed.
- The fault has to be confined.
- The functionality has to be taken over by another component. If the functionality of the erroneous component is not replaced completely then the system operates in a degraded mode (graceful degradation). The functionality of the erroneous component can be taken over by a redundant component or another component that changes its purpose for this action.

The premise for the second case is that an error recovery has taken place. After the disappearance of the error within the component two steps may occur:

- The component is reintegrated in the system.
- The component takes over its original functionality

The reintegration of repaired components could enhance the survival probability of the system significantly.

In general, two technical implementations are possible:

- Redundant components: Two or more components with the same functionality are installed in a cold- or hot-stand-by manner. When an error is detected in one component, this component is disabled and one of the remaining components takes over its duty. Fail-silent components disable themselves, non-fail-silent components must be disabled by a supervisor. In a cold-standby system this supervisor is also responsible for activating the redundant component.
- Components that can change their purpose (functional alternatives [SCK04]): In case of an error in one component the functionality is relocated to another component. The component that takes over the functionality does not execute its original function anymore. This is only an appropriate solution for control components that are not connected to special hardware. There are already prototypes in the telematics domain of the function relocation, e.g. when the on-board display of navigation system fails to work, an enhanced audio navigation assistant can be switched on to provide the driver a degraded navigation service. In an automotive safety electronics, however, it will be a challenging task to undertake the relocation in real-time. One example for these technical realizations could be that the control of a SafeSteering can take over part of stabilizing functionalities by providing more dynamic inference with steering when the braking management System goes to a degraded mode due to an error.

After having discussed about the constraints and guidelines about dynamic reconfiguration, in the following concrete functional design of dynamic reconfiguration will be given:

The requirements of deterministic behavior and dynamic reconfiguration seem to be contradictory at the first glance. But a closer look shows that a system that is reconfigured during runtime can still be deterministic. The reason for this is that we have a fault hypothesis containing errors to be tolerated. Thus we know which errors can occur and can therefore define statically how the FMF should react to the errors. We can define e.g. several schedules or configurations which based on the detected errors are switched on the fly (see Figure 9-33).

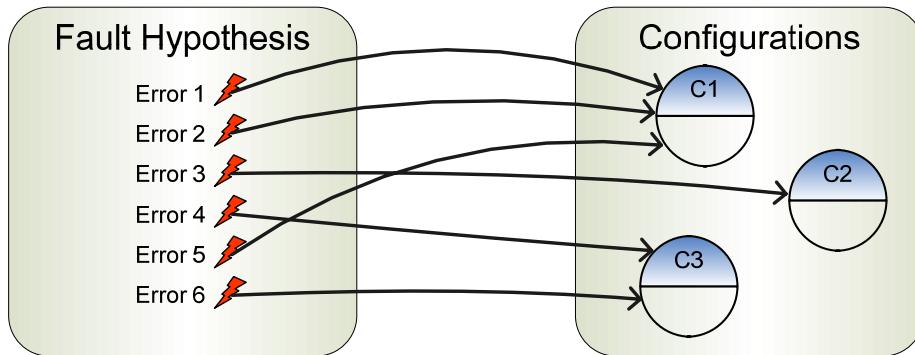


Figure 9-33: Mapping of Errors to Configurations

Based on the mapping concepts shown in Figure 9-33, dynamic reconfiguration can be divided into

- Function/runnable level: passivation or termination of faulty functions/runnables
- Task level: passivation or termination of faulty tasks
- Application level: passivation or termination of faulty application on the same ECUs or applications distributed on remote ECUs.
- ECU level: hardware reset of ECUs.

In order to perform the above reconfigurations, the OSEK-compatible OS as specified in Subsection 9.3.2.3.1 should be extended with the following functions:

Kill OS-Application	The operating system frees all system objects, e.g. kill tasks, disable counters and interrupts, etc., which are associated to the OS-Application. OS-Application (refer to Subsection 9.3.2.3.1 for detailed information) and internal variables are potentially left in an undefined state.
Kill OS-Object	The OS terminates an OS-Object (e.g. task or ISR) and releases all its held OSEK resources and re-enables all interrupts disabled by it.
Stop an OS-Application	Activate a task in the OS-Application and allow this task to stop (in a consistent manner) activities of the OS-Application.
(Re-) Start an OS-Application	An OS-Application is restarted after the self-termination (or being killed by the OS because of a protection error). If the OS activates the OS-Application it uses the configured restart task.

Function/runnable level reconfiguration

Function level reconfiguration is an indirect reconfiguration mechanism; it means that the FMF does not act directly on the function but provides the information whether the function should be activated or not. The term function is used here in an abstract way, it could be a runnable or any other element which can be activated or deactivated. The function level reconfiguration keeps track of certain events that have occurred which should prevent a function from being executed. For example, if a sensor signal is missing or faulty a function using this value should not be executed. The reconfiguration service will govern the status regarding whether a function is allowed to run or not, but it is the responsibility of the application to check the status before calling the function. In case the function is inhibited, the application may choose some backup strategy but this does not concern the FMF.

To manage the inhibition status an *inhibition mask* is used where bit X indicates whether event X has occurred or not. This mask is then compared with the *inhibition conditions* that have been specified for the function. These describe which event combinations that should cause the function to be inhibited. The reconfiguration service gets the event notices from the supervision.

To enable function level reconfiguration the following properties must be specified for each function:

- Function/runnable ID
- IDs of the events that should result in an update of the inhibition status
- Inhibition conditions

Task level reconfiguration

For task level reconfiguration the FMF offers the following possibilities:

- Termination of faulty tasks (and chained task) with help of OS
- Deactivation/Activation of tasks (and chained task) with help of OS
- Restart of a task, i.e. the init-function of the task is called.

Task level reconfiguration is more powerful than function level reconfiguration and requires support from the OS. Moreover, since deactivating/activating tasks has a strong impact on the system behavior, it is required that all activation patterns and tasks are known beforehand in order to enable a dependability analysis. That is, all fault management strategies involving task reconfiguration must be specified off-line and it is only the supervision that is entitled to issue such reconfiguration commands.

Termination of faulty task (e.g. tasks, which violate the space and time partitioning, which have a hazard to other safety relevant tasks) refers to an immediate reconfiguration to kill the running task with OS. The resulted error-hook reports the error to the OS. OS terminates the task with *KillTask<TaskId>* to free all the resource allocated by the faulty task. In case that the output of the faulty task is awaited by the following task, the so-called chained task, and the following task should be deactivated as well.

Tasks may be either time-triggered or event-triggered. Time-triggered tasks are activated solely by the OS based on timer interrupts. Event-triggered tasks are activated by having an external entity (e.g., another task) calling *ActivateTask(<TaskId>)*. The external entity can also be an interrupt

routine, e.g., associated with a timer. The task type affects how the deactivation/activation is to be implemented by the reconfiguration service.

Deactivation of a task means that following invocations of the task will be disabled. However, if the task is currently running it will be allowed to finish compared with task termination. The deactivation involves modifying the timers for the task (if it is time-triggered) or disabling the events/calls that trigger the task (if it is event-triggered). For safety reasons a running task may not be killed by any other entity than itself or the OS. This is why the task deactivation activity does not involve killing the task. It might, however, be necessary to kill a task that has become corrupted and is, for example, stealing processing capacity from other tasks. Activation of a task means that the events or timers that trigger the task are enabled again. Activation may also require (re)initialization of the task, that is, the init-function of the task is called.

Restart of a task means that its init-function is called. Thus, each task must register their init-functions as callbacks with the FMF.

One critical point is if a task responsible for the acquisition of sensor signal or give the commands to actuator is involved in the task level reconfiguration, or if the output of the task could block some other chained tasks, a fail-safe mechanism should be provided to avoid deadlock or failure propagation. During the reconfiguration, some callback function can be triggered by the error-hook to provide the “service-not-available” or “inactive” information for the chained task consumer of the task output. Another possible mechanism is to provide the consumers of the task output with time out mechanisms or signal counter to switch into the fail-safe mode. Since the measures here are most application specific, no detailed discussion will be given in this thesis.

Although it is the supervision that issues the reconfiguration commands, the reconfiguration service keeps track of the current state in a manner similar to functional level reconfiguration. This enables applications (or the rest of the FMF) to check whether or not a task is active.

Application level reconfiguration

Strictly speaking, the application level reconfiguration does not differentiate from the task level reconfiguration significantly. As depicted in Figure 9-31, application, distributed on different ECUs can be reconfigured with distributed FMF on each node. The pre-conditions for the application level reconfiguration are the synchronized configured fault-treatment on the different nodes, which requires a time-triggered communication system for the synchronization the reconfiguration and enough bandwidth to meet the real-time requirements. A concrete design and prototyping of the application level reconfiguration will be shown in Subsection 10.3.1.

ECU level reconfiguration

Suppose that one or some of the most safety relevant applications on one ECU are identified as faulty, in this case, a reconfiguration to fail-degradation and fail-safe will be executed immediately. If there is the need to heal the faulty ECU in the same drive cycle, the most direct and simple way is to restart ECU. Since there are quite a few fault handling strategies linked with the ECU reset, the reset demands should be arbitrated and performed with a few pre-conditions and pre-reset procedures.

In FMF, the reset manager is responsible for arbitrating the reset demands and calling the necessary system/OS service to perform a reset. Note that the arbitration between requests has to

be defined in the reset manager configuration. In order to guarantee a deterministic manner also in reset, the arbitration should be carried out with preconfigured look-up table or state-machine.

The reset manager is also responsible for keeping track of the reason why a reset occurred. The reset manager provides the following services:

- *FMF_RMInit*: This service is called by the supervision scheduler during start-up.
- *FMF_RMSHutdown*: service is called by the supervision scheduler during shutdown. It takes as parameter the shutdown reason passed to *FMF_Shutdown*.
- *FMF_RMGetResetStatus*: This service may be called by any FTU at the start-up phase to determine the cause the shutdown.
- *FMF_RMRequestReset*: This service may be used by any FTU to request a reset of the ECU, for example for fault correction or reconfiguration purposes. A parameter identifies the request in order to allow the reset manager to perform arbitration between different multiple and maybe conflicting requests.
- *FMF_RMInhibitReset*: This service may be used by any FTU to ask the reset manager to inhibit resets requested through the *FMF_RMRequestReset* service. A parameter identifies the enable/disable request in order to allow the reset manager to perform arbitration between different multiple and maybe conflicting requests.

9.3.4 Dependability software services for gateway

Gateway, as mentioned in Subsection 3.3 and Chapter 8 about hardware architecture, is commonly applied to connect different in-vehicle domains and external of vehicles. The general definition of gateway is a functional unit that interconnects two computer networks with different network architectures, for the ISS-safety systems, gateway is an ECU, which receives, process, save or forward data from one protocol to the other. Typical examples are ECUs, which process the sensor data and forward the signal to other in-vehicle network nodes or forwarding signals from one in-vehicle communication network to the other, e.g. LIN to CAN, CAN to FlexRay, Cabin-CAN to chassis/powertrain CAN, etc.

The ISS gateway services as depicted in Subsection 9.2 are summarized as following [GTS06]:

- Routing Service
- Network Address Translation: The Network Address Translation Service allows the mapping of addresses from one domain to addresses of the other domain.
- Communication Service with security and integrity
- Firewall Service
 - Authentication Service
 - Cryptography Service
 - Access Control List

As AUTOSAR provides basic inter-domain communication mechanisms, the AUTOSAR SW-architecture, incl. the AUTOSAR PDU Router for the routing between different networks, is taken as a basis for the gateway services discussed here.

As stated in definition of dependability in Subsection 1.2, dependability includes aspect of safety, integrity and confidentiality. Although gateway services are not the focus of this dissertation, in the following, the dependability issues (as issue 3 and 4 of the list, bold font) of gateway will be discussed briefly.

Communication service with security and integrity

Gateway dependable communication service here includes read, write (store) and forward of the signals. By both processes of read and write, plausibility check as well as data integrity check (e.g. with CRC) should be performed.

Signal routing includes usually unpack from original protocol, assemble/segmentation and pack in the new protocol. There are two critical aspects here: timing requirement and data integrity. For the end to end data transmission via gateway, the transmission time on the network is more or less limited with network bandwidth, so that delay time on the gateway can become the bottle neck of the overall delay. The layered software architecture should be optimized for the data processing and forwarding. In order to guarantee the data integrity, end-to-end application CRC as specified in Subsection 9.3.1.3 can be applied. In order to accelerate the process of data forwarding, the signal frame CRC might not be checked again, since in case of any fault detection, the signal should be still forwarded to the receiver with a flag for the fault identification. In order to protect the information of assemble/segmentation, a CRC to the whole block of message can be added as a signature of the gateway at the end of the pay-load.

Security in telematics gateway

One of the most important features of ISS-applications is the interconnection with telematics service provider. As some ISS applications require communication with external agents through a wireless link, like other vehicles for cooperative driving, or with the vehicles environment and infrastructure for transmission of floating car data and reception of traffic information, the gateways are needed to provide suited communication links. Security in telematics gateway is an important step in the integration of telematics services for safety systems. As specified in the list, however, the security issue here does not differentiate a lot from the security topic in the IT-industry, esp. with the rapidly increasing computation power of embedded system.

9.4 Configuration of dependability software services

Interfaces between basic software services and dependability software services, which are specified in the EASIS projects, are summarized and depicted in Figure 9-34. The dependability software services discussed in this dissertation are:

- Dependability RTOS – Subsection 9.3.2.2 and 9.3.2.3.1
- Fault tolerant communication services – Subsection 9.3.1.2 and 9.3.1.3
- Agreement Protocol – Subsection 9.3.1.4
- Software Watchdog – Subsection 9.3.2.3.2
- Fault Management Framework – Subsection 9.3.3.1
- Dependability software services for gateway – Subsection 9.3.4

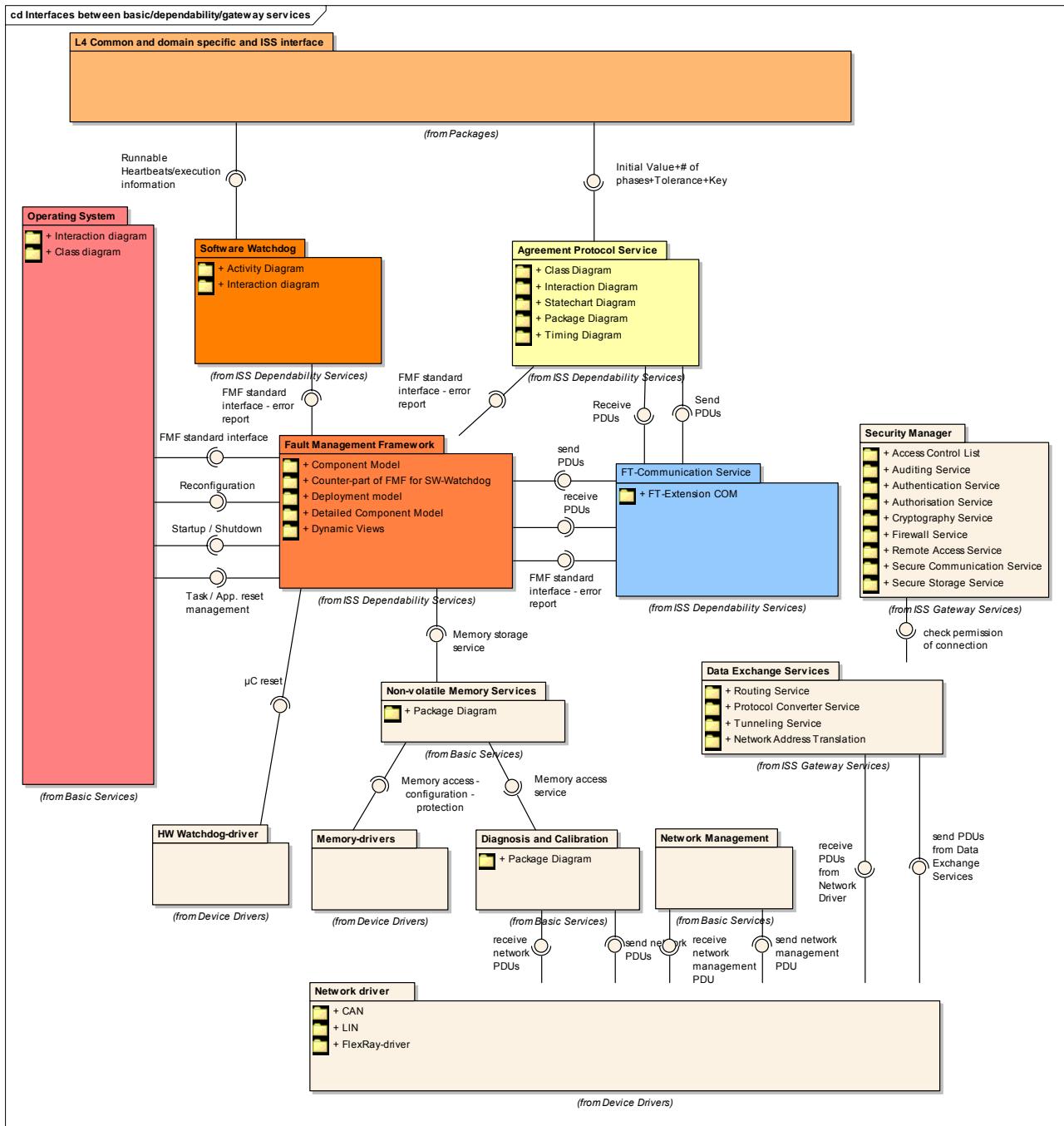


Figure 9-34: Interface of basic, dependability and gateway software services

RTOS is needed for each embedded electronic, the extended features introduced here can be categorized in different service classes as an add-on module to the core.

- Class 1: OSEK-compatible RTOS core with scheduling, basic resource management etc
- Class 2: Support for MMU and MPU (space partitioning)
- Class 3: Support for extended protected application services in time partitioning (execution time monitoring)
- Class 4: support for Fault Management Framework in dynamic reconfiguration e.g. termination, passivation of faulty tasks and task-groups

- Class 5: Support of full space partitioning and time partitioning services (e.g. runnable-call, periphery access protection, etc.)

It is to mention here that Class 2 is needed for the following two classes, while Class 3 and Class 4 are more or less independent from each other.

In a similar way the communication services can be divided into the different classes as well:

- Class 1: OSEK compatible COM with support to different in-vehicle networks
- Class 2: Fault tolerant COM with interface to application CRC
- Class 3: Support for redundancies and interfaces for additional fault tolerance mechanisms
- Class 4: Integrated fault tolerant communication services with Agreement Protocol

Class 1 and Class 2 are relative generic without preference for the communication networks, while the Class 3 and Class 4 are only required by the certain safety application with additional support.

The two main services provided by Software Watchdog, Aliveness Monitoring and Control Flow Checking, are also quite independent from each other. They can be configured to the needs of applications, the monitoring objects and threshold/counter borders can be configured in the look-up table.

The 4 sub-components of Fault Management Framework can be configured individually as well. The configuration varies from the simplest case as a fault handler to the complex variant, coordinated system wide reconfiguration with help of SW-Watchdog and OS. The reconfiguration part is the most complex component, while the reconfiguration strategy, as highly application dependent, could be chosen based on the framework defined here.

Dependability software services for gateway are only needed for gateway ECUs, the security issue is of course only interesting for the links with external communication.

9.5 Conclusion of the dependable software platform

In this chapter, started with a benchmarking with IT-industry, a dependable software platform is specified as an enable lever to improve the system dependability. Focused on the three aspects of Integrated Safety System, the communication (including gateway), the integration of application on one hardware platform and the fault treatment are discussed. Last but not least, the dependability software services are classified in different classes for the configuration and integration.

10 Prototyping and validation of the concepts

Before we can explore the prototyping and validation of the concepts for the future automotive electronic architecture in detail, a brief overview of the development process, methodology of the validation activities is given in the following subsection.

10.1 Introduction to the architecture validator

10.1.1 Prototyping approaches and validation process

As mentioned in Subsection 3.5.2, state of the art in the development of in-vehicle electronics is the model and simulation based development process, in which a sub-system is developed and tested in a pure software environment with the help of simulation tools (Model-in-the-Loop and Software-in-the-Loop test). The implementation of software on a particular hardware platform will only be initiated after a successful test of the functionalities with the help of simulation models.

Here this approach was adopted and extended – using rapid prototyping systems as real-time execution platforms, which allow for short design cycles while evaluating effects of different realization choices including the flexible variation of hardware setups.

Due to the networked architecture, the development process needed to cover system and node level design as depicted in Figure 10-1. The tools used for early development and validation are mainly MATLAB/Simulink together with Real-Time Interface Block-sets and experimentation tools, which enabled the validation of the dependability software services on rapid prototyping platforms with FlexRay interfaces from dSPACE [Eck06].

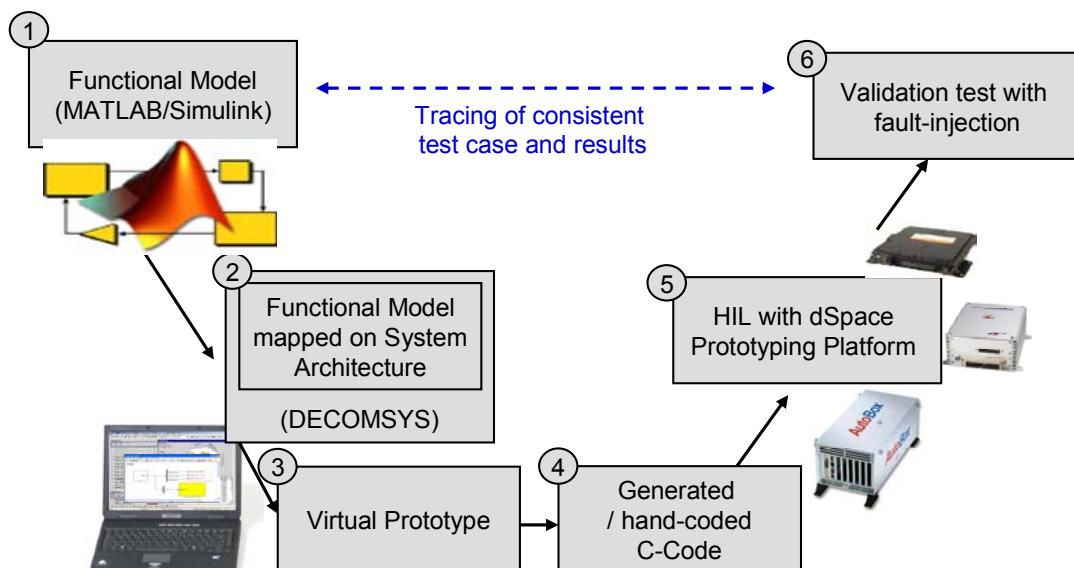


Figure 10-1: Tool chain and development process

As shown in Figure 10-1, after requirement analysis of the safety applications,

- The first step the system functional design is initiated by building and modeling the whole system including the simulation environment with MATLAB/Simulink.
- The dependability software services are prototyped and simulated on a PC as a virtual prototype in the second step.
- Based on the principle of fault injection, the virtual prototype is evaluated as a front-loading in the third step.
- In the fourth step, based on the knowledge gained from the virtual prototype and other automotive constraints such as resource and timing requirements, the virtual prototype of the dependability software services and the modeled safety application will be mapped onto tasks and scheduled on the system architecture. For the rapid-prototyping platform C-code can be generated automatically. For the evaluation-board source-code can be manually implemented with the development environment of the certain microcontroller.
- In the fifth step, source codes will then be compiled and loaded onto the rapid prototyping platform using AutoBox or MicroAutoBox [Eck06] from dSPACE or evaluation-board. Dependability software services and concepts are evaluated in the real hardware environment and tested against the requirement specification, by injecting the faults which are supposed to be tolerated by the dependable software platform. The consistent evaluation cases and validation concept will be applied through the virtual prototype, rapid-prototyping and on evaluation-board, thus the traceability of the testing results to the design specification through different platforms and environments can be guaranteed.

10.1.2 Architecture design of the validator

As mentioned in Chapter 5, the validation of concepts in this dissertation took place in the EASIS WT1.3 Prototypical Implementation [VAL06] and WP 5 – WT5.1 Architecture Validator [VAL06] and WT 5.2 Process Validator (Commercial Vehicle validator). The EASIS architecture validator [VAL06] is focused on the prototyping and validation of some of the most relevant properties of the EASIS architecture both in hardware and software, including fault tolerant hardware, dependability software services and telematics services etc.

The implemented EASIS architecture validator (as depicted in Figure 10-2) incorporates the following components:

- Fault tolerant communication with dual-channel FlexRay network
- Fault tolerant ECUs with dual duplex architecture
- Dependability software services providing dependability software services, upon which further applications can be built, including:
 - Agreement Protocol
 - Software Watchdog
 - Fault Management Framework
 - Dual-duplex fault tolerant signal processing

- Telematics gateway between internal buses (dual channel FlexRay inter-connected with two active stars and CAN) and external Telematics network (TCP/IP) with protocol conversion and security services.

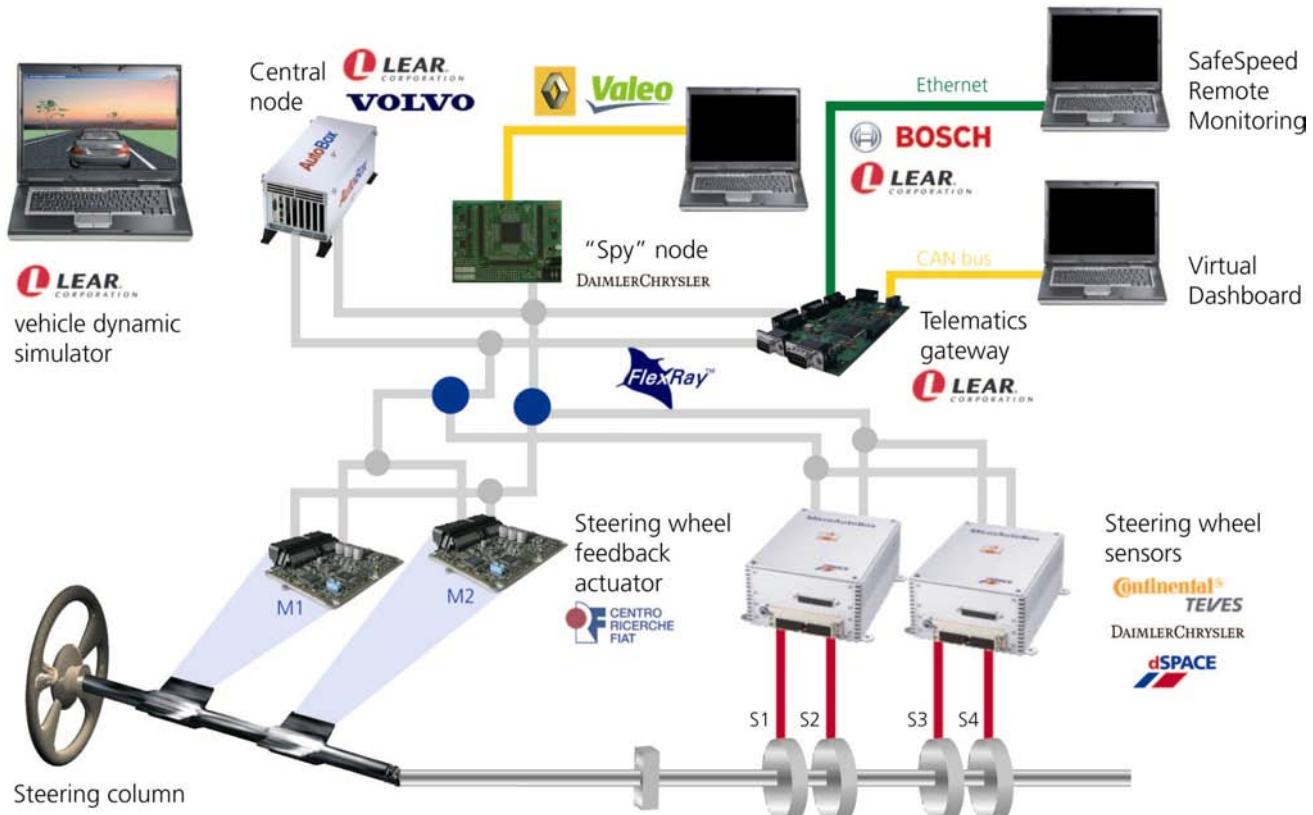


Figure 10-2: System topology of EASIS architecture validator [AfS07]

The most important hardware nodes are the Sensor Node (steering wheel sensors), Actuator Node (steering wheel force feedback actuator), Central Node, Spy Node and Telematics Gateway.

In order to show the behavior of EASIS architecture from the application's point-of view, the EASIS validator includes a safety-critical function (steer-by-wire) and three demonstrative integrated safety applications:

- SafeLane – a lane departure warning system
- SafeSpeed – limitation of vehicle speed by an external commanded value and remote monitoring using Internet browser.
- SafeLight – an enhanced adaptive lighting system with help of vehicle dynamic and environment information.

The EASIS architecture validator also includes several interactive graphical interfaces to show, in real time, the behavior of the EASIS architecture and its resilience in front of faults. It is possible to inject faults (both hardware and software) on the system and monitor the system behavior including a 3D vehicle dynamics simulation.

10.2 Validation of hardware architectures for ISS

In the following, the validation process and results of the most important safety concepts of hardware architecture in Chapter 8 will be described.

System topology with backbone concept

In the three system topologies discussed in Subsection 8.1, we chose the backbone concept for the validator, since the integrated safety applications based on the steer-by-wire technologies require fault tolerant communication with high-bandwidth. Compared with the Figure 8-14, the system topology of the validator is almost the right part with the dual-channel chassis FlexRay and telematics gateway. As mentioned in Subsection 3.3.4.2, the active stars provide better fault tolerance than the passive star topology of CAN due to the detection of short cuts and deactivation of branches in the active star.

Result: The backbone concept with FlexRay enables integration of ISS-communication with large amount of data transfer. The safety integrity requirement of communication system can be reached without any additional safety concept.

Distribution of ISS-application to the HW-architecture

The three high level ISS-application as well as two basis functions of Steering Wheel Sensing and Steering Wheel Feedback were mapped to the HW-architecture. The guidelines about portioning and mapping of ISS-applications in Subsection 8.4.1 are considered here.

1. According to rule 1, the actuator control with steering wheel feedback is mapped to dual-duplex ECUs directly connected with the electronic motor.
2. Since the ECUs for actuator control have also FlexRay interface, the sensor for vehicle speed and steering wheel angle can be placed freely.
3. FlexRay transfer rate was configured to 5 Mbit/s, no limitation of bandwidth is needed to be considered.
4. In the validator, the signals of speed and steering wheel angle are distributed with FlexRay. Steering wheel angle, for example, is sensed redundantly in a dual-duplex manner. The redundant signals are checked and voted for its plausibility in the value and time domain before they are sent to the bus.
5. The high level application SafeLane and SafeSpeed are integrated together on the Central Node, since they are of the same safety integrity level and the hardware platform AutoBox provides enough resource for them. Dependability software services facing the challenges of application integration were also mapped to the Central Node.

Design concept of communication networks

There are mainly 3 communication networks integrated in the validator: CAN and FlexRay are mainly used for the vehicle dynamic control while Ethernet is applied to represent the telematics communication from extern. CAN is applied for the coupling between physical sensors and sensor

node, and also monitoring/fault injection (see Subsection 10.3.3). While the latter use-case is not safety relevant, safety concept mentioned in Subsection 8.2 is implemented to tolerate the timing drift of sensors.

Result: Switches for fault injection were built into the validator to emulate DC fault model of the network. FlexRay Scheduling requires a high amount of front-loaded design work, since all the signals have their fixed place in the communication schedule. OEM, together with Teer-1 suppliers should spend more effort in the system requirement engineering and design phase to fix the communication matrix. This effort, however, is paid back upon the system integration, the FlexRay communication was integrated almost like plug-in & play and works very reliable and.

Dual-duplex architecture

Dual-duplex architecture can be found both in the safety concepts of the actuator and sensor-coupling in the validator as shown in Figure 10-3. The steering wheel angle is the one of the most important inputs of steer-by-wire function. The sensing function, with the safety integrity of ASIL-D, should be redundant and independent in the path of signal processing. Thus the two sensor node, each connected with two physical steering angle sensors, can work completely independently in a fail-operational mode. The exchange and voting of the angle is carried out with FlexRay.

The dual-duplex actuator has architecture of 1oo2D-system, with independent power supply and communication path to both FlexRay channel A and channel B. The application software is partitioned between the microcontroller and the DSC in such a way that the DSC practically acts as an actuation co-processor, providing the management of the electric motor at low level, while the main microcontroller handles the actual application software and provides interface to the rest of the system and to the end user for calibration and diagnosis (for more details see Appendix 1).

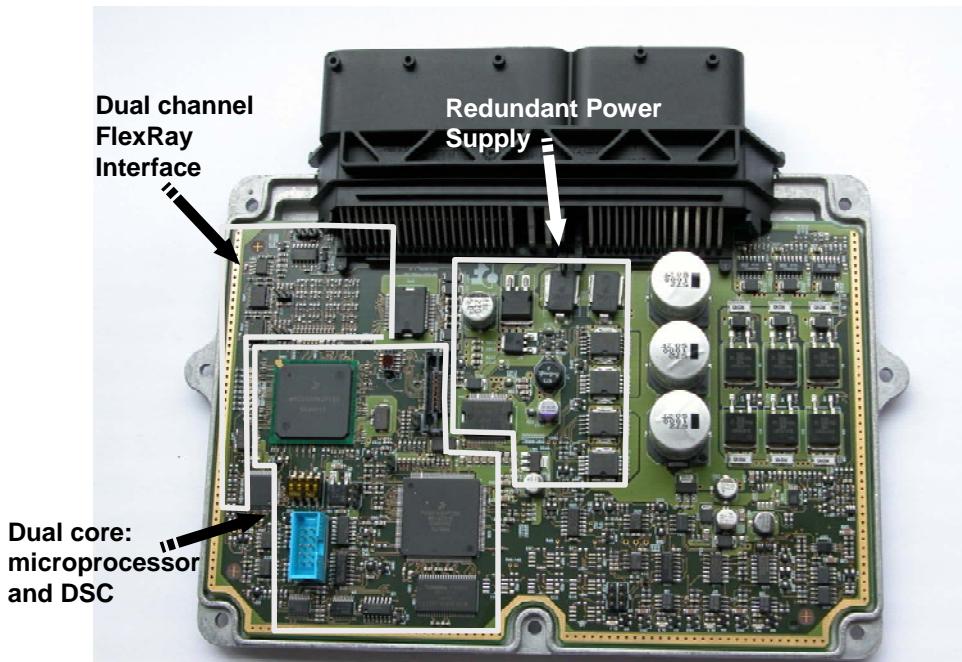


Figure 10-3: Fail-safe unit of dual-duplex actuator ECU

Result: The dual-duplex architecture guarantees enough safety integrity, which is required for the fail-operational behavior.

10.3 Validation of dependability software platform and services

The validation of the dependability software platform as shown in Subsection 9.2 is performed during the design and integration of ISS-application with the underlying software platform. Design of the software architecture on the Central Node, Sensor Node and Spy Node followed strictly the layered architecture. The smooth integration of applications and dependability software services from different consortium partners in very short period demonstrates the effectiveness of the approach.

Based on the requirements and fault hypothesis in Chapter 6, fault injection is applied to validate the dependability software services along with exemplary fault handling strategies from the praxis. Before we will go into the details about the validation of each software services, the principle of fault injection is given as following:

Typically fault injection can be divided into 3 main types ([Hed01] and [Ehr03]), physical fault injection, software based fault injection and fault injection in simulation model. For the validation of dependability software services, because physical fault injection (EMC, circuits stuck-open, open or high impedance outputs) are inflexible and not versatile enough in cope of different fault types, it was only applied in the validation phase for the validation on the communication and hardware architecture. For the validation of dependability software services, software based fault injection and fault injection in simulation model are intensively applied.

Following the basic theory of fault tolerance as shown in Subsection 2.2, there exist two basic approaches: fault injection and error injection. Fault injection simulates software design faults by targeting the code. Here the injection considers the syntax of the software to modify it in various ways with the goal of replacing existing code with new code that is semantically different [Voa98]. This “code mutation” can be performed at the source code level before compilation if the source code is available. Error injection, called “data-state mutation” in [Voa98], targets the state of the program to simulate fault manifestations. Actual state injection can be performed by modifying the data of a program using any of various available mechanisms.

The relationship between faults and errors is depicted in Figure 10-4. We can see that one fault can result in several errors (common mode) and on the other hand several faults can cause the same error (same syndrome). Generally speaking there exist much more faults than errors, e.g. a time violation of a task may have many different reasons (e.g. corrupted scheduling table, corrupted memory or bugs in the tasks).

As long as all the errors can be tolerated, all the faults in the fault hypothesis can be tolerated with the dependability services as well. Here it is to mention that no 100% complete mapping between all the possible faults and errors are needed to be identified. Another experience is that errors are generally easier to be injected than faults, e.g. emulation of memory faults to influence the control flow of runnables is more difficult to implement than the direct error injects in the control flow with manipulation of program counter. Because of the reasons mentioned above, the validation of the dependability software services in the following subsections is based mainly on the “error injection”, strictly speaking.

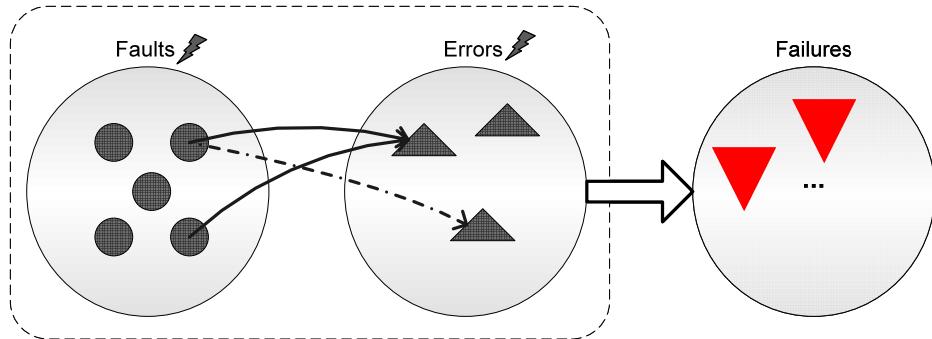


Figure 10-4: Mapping between faults and errors

10.3.1 Prototyping and validation of Agreement Protocol

As discussed in Subsection 9.3.1.4, Signed Messages Protocol was chosen as a basic concept, upon which an Agreement Protocol Service was designed and prototyped, while the following requirements and constraints are considered:

Firstly, the Agreement Protocol Service executed on a node must deliver not only an agreed value, but also a state vector, which includes the state of each participating node. This state is gained from last agreement process from the point of view of the node that generated it. The state vector could contain useful information in case of a failure to identify the faulty nodes in a system.

Moreover, the signature mechanism should not be as complex as for IT applications, since efficiency and short execution times are required. Therefore a simpler and thus more efficient signature mechanism was designed and integrated in the Agreement Protocol Service.

Usually, fault-tolerant applications have backup system states, which are applied when faulty components or other abnormal states are detected. In this case, all the distributed safety components of the application must be consistently informed of this change in order to take synchronized and coordinated fault treatment measures. Therefore, the SafeLane application in the EASIS validator has, besides the “operational mode”, a “faulty sensor mode” and a “failure mode”. Since the application includes the central node and the two fail-safe sensor nodes. Each fail-safe node is equipped with two physical steering wheel angle sensors [SWS03], thus they implemented a fail-operational sensor node in a dual-duplex structure. The global application state is computed from the single states of these nodes.

As shown in Figure 10-5, the Agreement Protocol Service was embedded and validated on the three nodes (Central Node and two steering wheel sensor nodes) and is cyclically executed in order to maintain consistency among them with respect to the global state information of SafeLane. The private value of each node is its own state. The state of each node is determined, as depicted in Figure 10-5, by a state machine as demonstrated in Figure 10-6, whose transitions depend on local node information, as well as the global application state as an output of the Agreement Protocol Service. The state information is exchanged between the nodes over FlexRay which is not only reliable and deterministic, but also exhibits the required cyclic behavior. Upon the execution of the agreement protocol, the non-faulty nodes consistently build a vector consisting of the different nodes states. Based on this vector, each node consistently computes the global application state by applying a specific decision function designed for the SafeLane application.

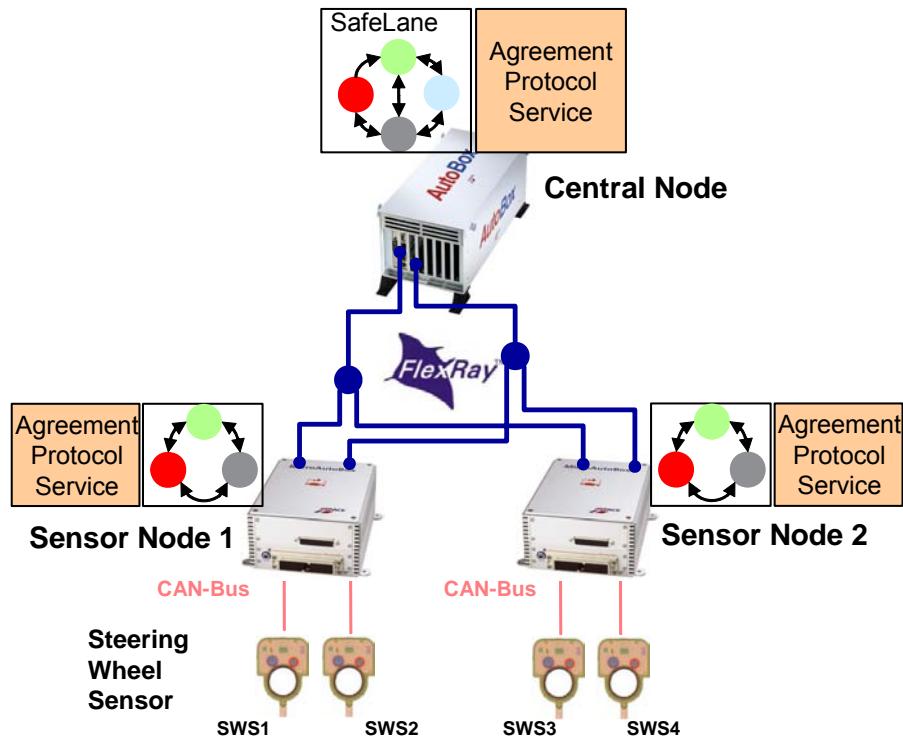


Figure 10-5: Outline of system architecture for the validation of Agreement Protocol Service

Hence, the use of the Agreement Protocol Service guarantees a reliable and deterministic global state transition among the three nodes so that an agreed and safe state can be maintained for the SafeLane application.

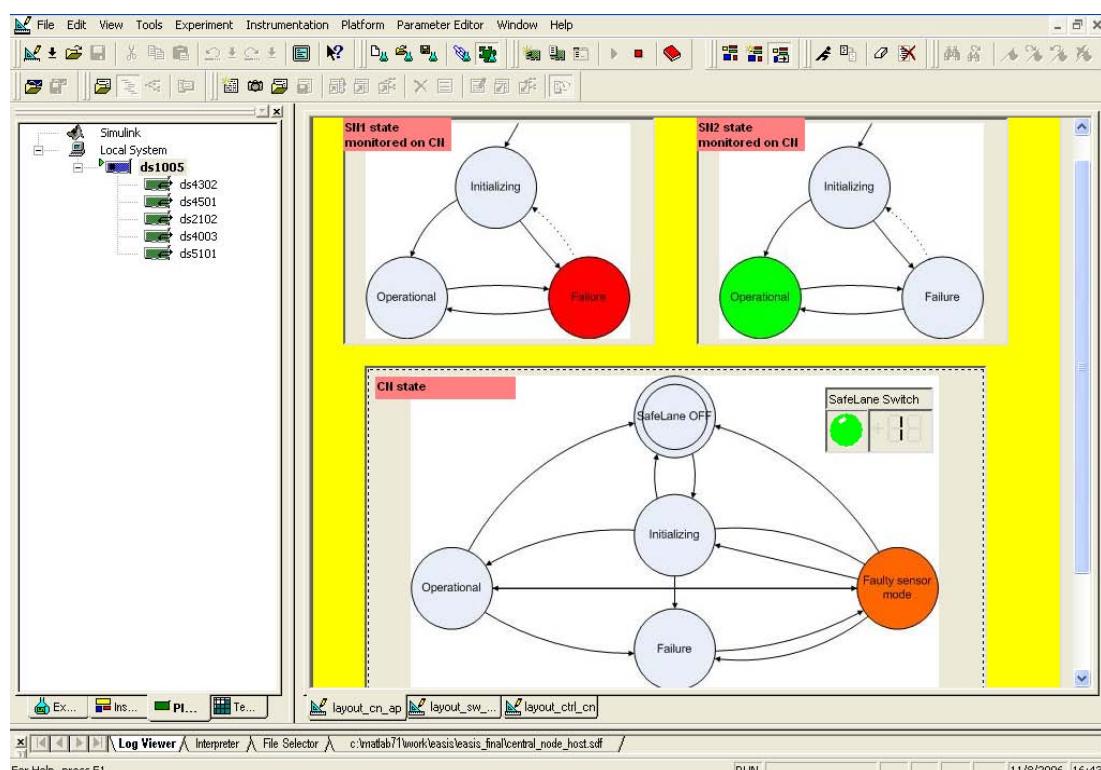


Figure 10-6: User interface for agreement protocol validation

The validation of Agreement Protocol follows the principle of fault injection according to the fault types defined in the design phase. Different faults like FlexRay communication fault, Byzantine Fault whereby inconsistent data is exchanged between the nodes and memory fault of inconsistent data read, etc. are injected to the validator. The faults can be injected with hardware failure or emulated with software using dSPACE ControlDesk. Figure 10-6 shows the user interface with the state machines of each node implementing the agreement protocol changing in real time.

10.3.2 Prototyping and validation of Software Watchdog

The prototyping and validation of Software Watchdog follows the complete steps as shown in Figure 10-1 from virtual prototype to evaluation-board, which is explained in depth as following:

Modeling, simulation and prototyping of the Software Watchdog

For the modeling of task dispatching and program flow of runnables in OSEK, Stateflow in Matlab/Simulink was applied. Stateflow is a design and development tool used for modeling complex system behavior based on finite state machines. Runnables are modeled with function-call subsystems and triggered by events sent by Stateflow in a defined execution sequence. A function-call subsystem is a block in Matlab/Simulink which can be invoked as a function by another block. For instance, as illustrated in Figure 10-7, the application SafeSpeed can be divided into three runnables: sensor value reading in *GetSensorValue*, the control algorithm in *SAFE_CC_process* and setting of the actuator in *Speed_process*. These are triggered as function-call subsystems by the Stateflow chart *SafeSpeed*, in which the execution sequence of runnables is implemented. To indicate the aliveness of the runnables, further function-call subsystems to simulate the glue code are also implemented, which report the execution of the runnables.

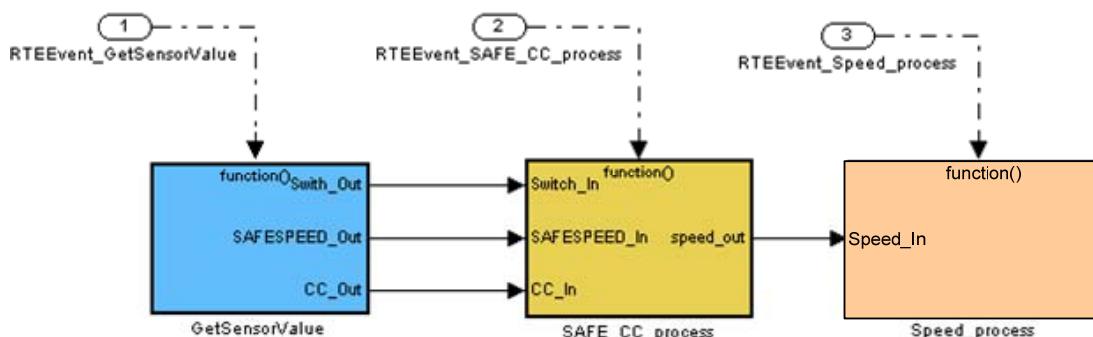


Figure 10-7: Modeling of runnables and program flow in SafeSpeed

The time-triggered behavior of the heartbeat monitoring unit and task state indication unit was modeled with time counters. Thus, in order to simulate the mechanism of task scheduling with different periods in the operating system, different time counters can be assigned to the Stateflow charts. On the other hand, the program flow checking unit was modeled using an event-triggered Stateflow chart.

Evaluation of the Software Watchdog in EASIS validator

The evaluation of the Software Watchdog is performed based on the fault/error definition in the design phase. Since different faults can result in the same error, error injection is applied for the evaluation of the design and prototyping of the Software Watchdog. Such an approach has the advantage that the dependability requirements can be tested in a front-loading manner of system development. The concept can be validated independently from the specific fault-types. Faults, which are difficult to inject into the test bench or on-road test, can be relatively easily emulated with errors.

Here again Stateflow is used to manipulate the execution frequency and sequence of runnables by changing the timing parameter of runnables, manipulation of loop counters and building invalid execution branches, etc. The experiment environment ControlDesk provides the possibility to manipulate the data assigned to the timing parameter of runnables to the condition that determine the invalid execution branches in the runtime. Therefore, it is used to trigger the error injection during the execution of the applications and visualize the results as well.

By building different evaluation cases, the three chief functionalities of the Software Watchdog, i.e. the detection of the aliveness error, the arrival rate error and the program flow error, are successfully validated. The following screenshots demonstrate some of the evaluation cases generated by injecting heartbeat or program flow errors. The x-axes of each plot in the diagram indicate the time lapse, which has a scalar of 10ms. The y-axes indicate the value of the counter and number of detected error. In order to inject heartbeat errors, a time scalar is connected to a slider instrument to change the execution frequency. For example, Figure 10-8 shows the test with an injected aliveness error. Similar test with arrival rate error and control flow error were performed as well. The increase in the y-value in the last plot “AM Result” (Aliveness Monitoring Result) indicates the detection of the errors.

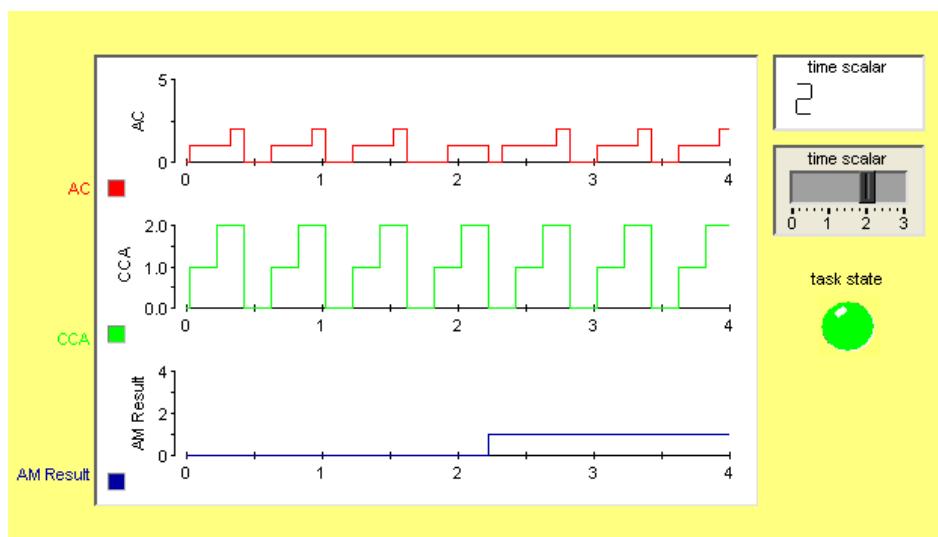


Figure 10-8: Test with injected aliveness error

Figure 10-9 shows the case in which the real cause of the erroneous state is identified through the collaboration of the units of the Software Watchdog. Here, the aliveness errors detected by the heartbeat monitoring unit are actually caused by program flow errors, which are reported with the

plot “PFC Result” (Program Flow Checking Result). After the detection of three program flow errors (which here is set as the threshold), the task state is set to “faulty”. Only one accumulated aliveness error is reported.

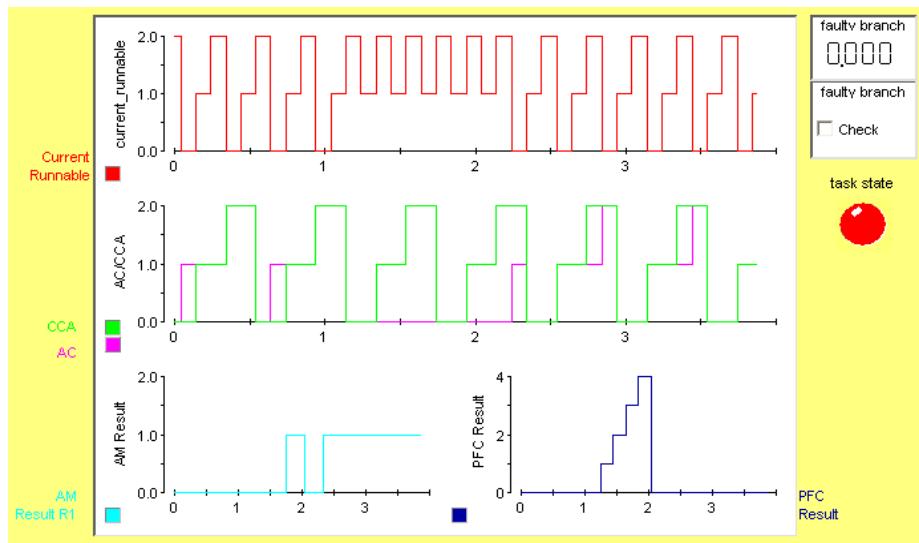


Figure 10-9: Collaboration of fault detection units

10.3.3 Prototyping and validation of Fault Management Framework

As well as other dependability software services, in the first step the system functional design of the FMF was initiated by building and modeling the whole system with Matlab/Simulink. The Fault Management Framework was prototyped and simulated on a PC as a virtual prototype in the second and third step. In the fourth step, based on the knowledge gained from the virtual prototype and other automotive constraints such as memory and timing requirements, an actual microcontroller was chosen here for the implementation and system validation. Individual hardware specific C-codes were generated or hand-coded. This version was integrated and compiled with the safety applications light control and other basic software modules, such as the operating system OSEK OS with an extended task scheduler. In the fifth step, the FMF along with the applications and environment were loaded onto the target platform and tested in a HIL. The development environment CodeWarrior from Freescale [CoW05] was applied for the debugging, compiling and linking. Thus, the virtual prototyping in MATLAB/Simulink eased the development of the FMF functionalities on the target hardware. More details about the prototyping of the FMF can be found in [FMF06] and [Spi07].

Validation with Fault Injection and Rest-bus-Simulation

The validation of the FMF followed the principle of fault injection. Generally speaking, fault injection can be possible at different system components and it depends on the test object. In this case we want to examine the designed functionality of the FMF. Therefore, one possibility would be to insert faults right into the FMF, that is to say, fictive faults can be injected and reported to the FMF to test the designed fault-treatment. On the other hand, the FMF works with application software

components and other dependability software services like the Software Watchdog mentioned above, so we can inject the faults by real faults as well. That is to say, the faults are injected on side of the application, which are detected by the Software Watchdog and then reported to the FMF. This second approach is more appropriate because it takes also the interaction between the FMF and other components into consideration.

In order to evaluate the fault management framework service, different evaluation cases must be considered. The main cases of fault injection are depicted in Figure 10-10.

1. In the first case, the fault injection will take place at application level (as demonstrated in Figure 10-10: faults are injected in light/rain sensors to result in the faults of light control).
2. In the second case, the Software Watchdog was applied to report errors to the FMF. Thus, we manipulate the scheduling table (as an extension to the OS) in order to trigger errors concerning runnables, which should then be detected by software watchdog and then reported to the FMF.

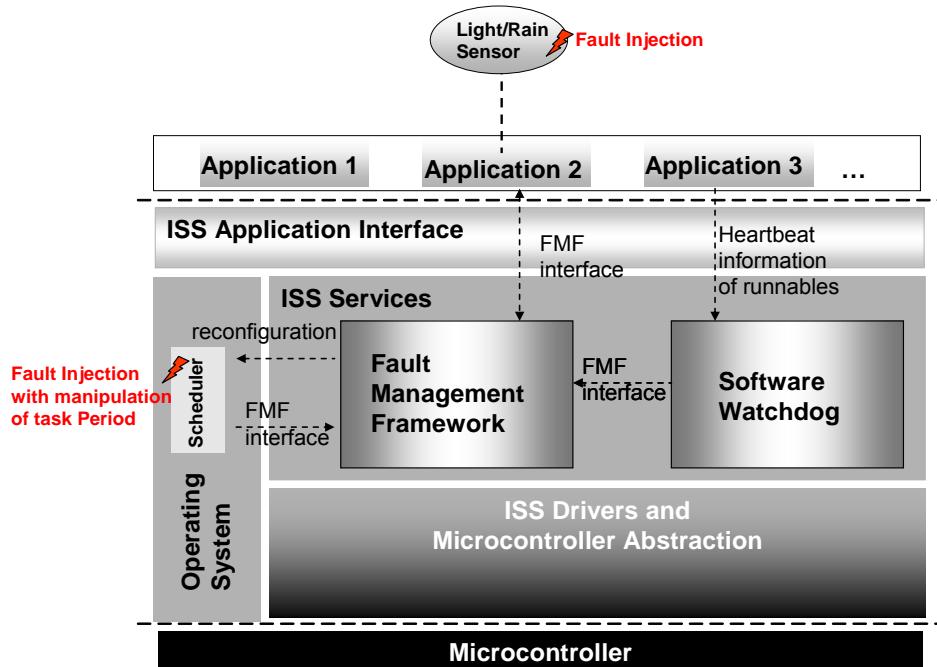


Figure 10-10: Evaluation cases with fault injection for FMF

In order to trigger fault injection and to display the evaluation results, a simulation and diagnosis tool (CANoe) was used, which was connected via a CAN card to the CAN interface of the evaluation board. The structure is depicted in the Figure 10-11.

The evaluation microcontroller S12XF was connected to the FlexRay network to collect the vehicle status information (speed, park, start...). A computer, running the tool CANoe, was linked to the ECU via CAN bus for simulation and demonstration effects, but also for rest-bus simulation. The graphical interface implemented with the CANoe panel editor was used to trigger fault injection on the one hand and to display the actual status of the ISS services, e.g. faulty state of a runnable, state of FMF and application sensors, etc.

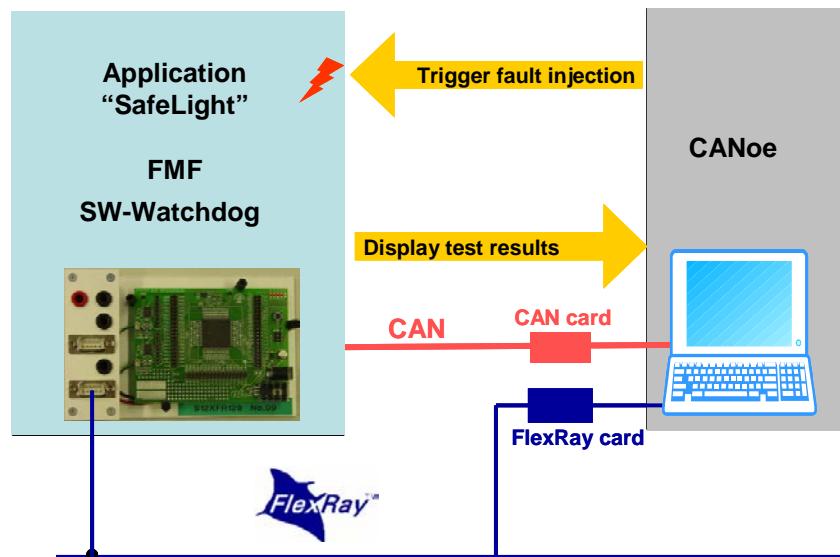


Figure 10-11: Fault injection triggered with CANoe to FMF

The rest-bus simulation here enabled a smooth integration in the EASIS overall architecture validator. The same test cases could be reused and traced in the fully networked Hardware-in-the-Loop test bench.

10.4 Validation of the ISS Engineering Process

As stated in Subsection 7.1 the two most important concepts of EASIS Engineering Process are

- Virtual Frontloading (virtual integration with structured test) and
- Correct by construction approach

These two principles are validated during the participation in EASIS work-package of the validator. Following the steps specified in the ISS Engineering Process, some of the most important validation results are summarized as following:

Development of Functional Analysis Architecture

The FAA-model, using model-based specification is depicted in Figure 10-12. As a hardware mapping and platform independent model, the FAA is validated with “model-in-the-loop” approach. The hardware independent FAA-model enables a clear specification of interface and data model, which can be easily transferred and further developed by the OEM and suppliers in a decentralized manner. Based on the safety analysis (e.g. preliminary hazard analysis) and fault types, the potential hazards of the system are simulated and injected into the simulation environment to the FAA-model with virtual front-loading.

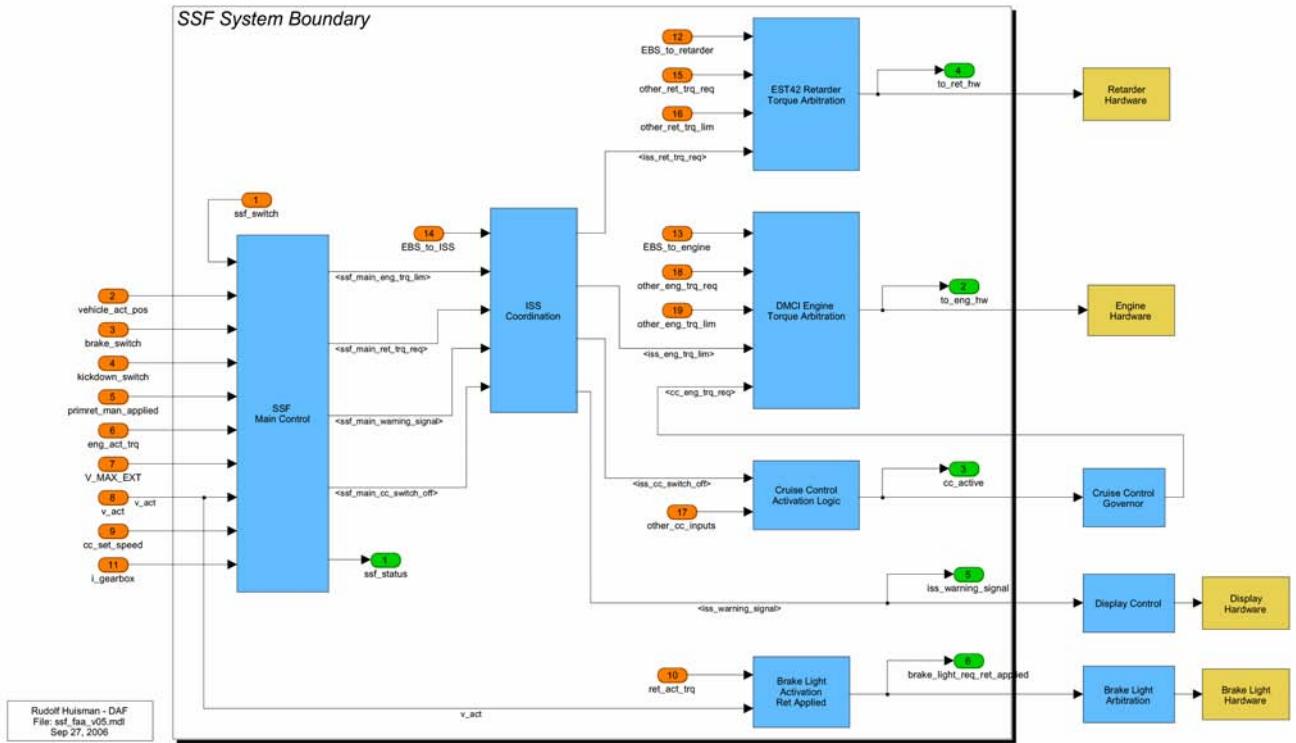


Figure 10-12: FAA-model of SafeSpeed

Development of basic Functional Design Architecture

By mapping the FAA-model to the HW-architecture and split the whole model to the subsystems, we can get the FDA-model as shown in Figure 10-13. The FDA-model here can be further supplement to the interfaces of basic software services. With the virtual integration of FDA with these HW-platform independent basic software services, the FDA-model can be validated with the "Software-in-the-loop". The same fault injections from the last step are recycled. With the suggested unified simulation environment and tool chains, the FDA-model can be exchanged simultaneously to any time point of the development between the development partners without any overhead customization. Moreover, software bugs and test scenario can be distributed and reproduced easily. The ISS EP enables a significant improvement of the interaction of partners to manage the complexity of ISS.

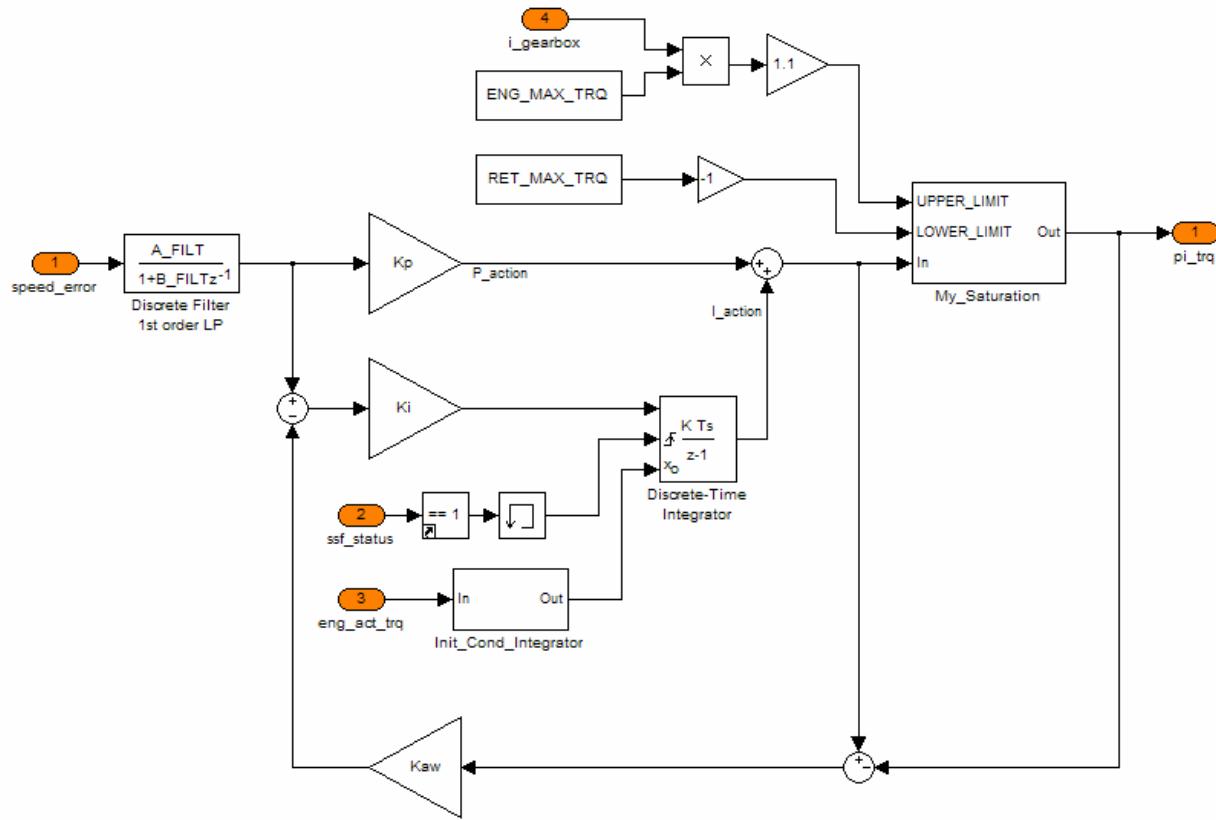


Figure 10-13: FDDA-model of the SpeedControl in SafeSpeed

Refinement of and validation of FDDA model with SiL-test

With the application of dependability architecture framework, the FDDA can be integrated to the layer software topology with the dependability software services. With the standardization of dependability software services, distributed test of safety relevant requirements (to which, the dependability services are designed for) can be carried out without costly shipment of special equipment. In the following Figure 10-14, we can see the FlexRay scheduling requires a global communication road-map at early development phase. Any changes of the scheduling will result in a chained reaction to the scheduling and resynchronization between application tasks and communication services on the other nodes. The consistent tool chain and standardized FT-Com services enables an uncomplicated exchange and update between the partners.

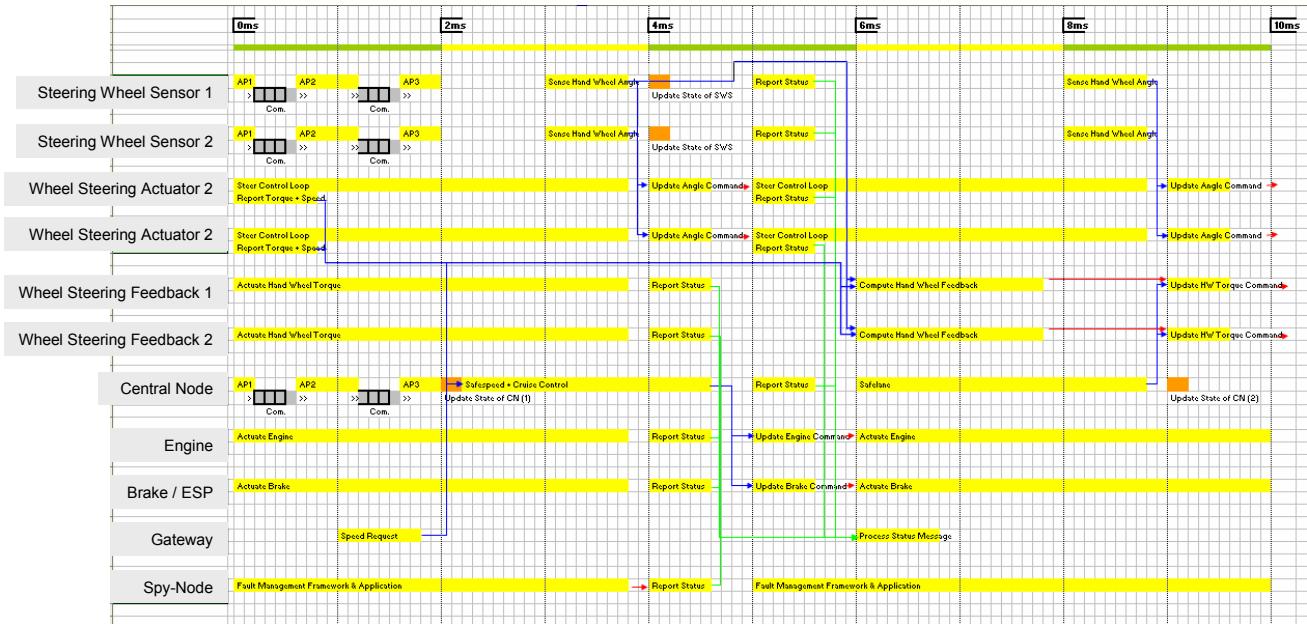


Figure 10-14: FlexRay communication scheduling in the validator

Experience in setting up of the validator, using a hybrid system, made up of the nodes and communication systems listed in ISS EP step 5.3, shows the flexibility and effectiveness the approach. The counter-part (remote nodes in the ISS) can be simulated and virtual integrated in the most cases. The definition of dependability architecture frameworks shortens the transition time between different development phases and prototyping platforms, e.g. PC-based simulation environment, RP to evaluation-board.

In all the integration and validation of total EASIS-validator was quite efficient. The physical assembling and system test of the different nodes is almost plug-in and play, no iterative design loop was required.

10.5 Evaluation and optimization of the concepts

Safety concepts proposed here are implemented with rapid-prototyping and serial near equipment, while the validation follows the principle of fault injection. During this validation phase, faults were injected into the system on hardware, application level as well as directly in the dependability software services. All injected faults are proved to be detected by the system correctly and the appropriated fault treatment strategies have been executed. However, it has to be noted, that only fault scenarios identified and defined in advance by the developers could be handled. As the aim of the work here is to validate the concepts, fault injection does not provide results about quantitative fault detection coverage and test coverage. Since the ISS-applications here are not directly from the serial projects, no confident performance benchmarking can be carried out.

As a summary, the results here with regard to hardware/software architecture and platform are here shown to be functioning properly, and they are also identified as being pertinent for upcoming Integrated Safety Systems. Therefore, services like those defined in the dependable platform have to find their way into real-world platform. A step towards bringing the results closer to being deployed in products will be to advocate the use of the results as input and inspiration for on-going projects concerning automotive software platforms, such as AUTOSAR.

10.6 Experience and findings from the prototyping and validation

In all, most of the concepts suggested in this thesis, including ISS Engineering Process, fault-tolerant hardware architecture and dependable software platform was prototyped and validated during the EASIS work-packages of validator. The prototyping and validation process here is not characterized with the traditional OEM/supplier (customer/vender) relationship, but with a more decentralized peer-to-peer relationship as project partners. It was quite challenging to make a master plan for the distributed development, such as to consolidate the design and prototyping approach and to control and synchronized the distributed activities. The common prototyping platforms as well as development environments despite of a few features (e.g. using interchange format of communication scheduling to synchronize the changes, migration from virtual prototype to the hardware platform) have contribute to the successful cooperation significantly.

11 Conclusion of the results, discussion and outlook

Future Integrated Safety System exhibits an ever higher system complexity with the integration of safety functions with different integrity levels. The dependability requirements as to the fault treat behavior, triggered by the industry norms, increase from fail-safe/fail-silent to fail-operational.

In the dissertation presented here, with the analysis of the key aspects of ISS, the requirements of future ISS are derived. With the assessment of these requirements with state of the art safety electronics as well as the current trends of the in-vehicle electronics and safety norms, a delta analysis for the concept development was carried out. To address these challenges and close the gap, an integrated dependability architecture framework for the ISS was specified in the three main levers: hardware, software and development process. As a conclusion the most important results and their implication are concluded as following.

11.1 Conclusion and implication of dependability architecture framework

Development process

ISS-applications are characterized with intensified interaction between applications of different safety integrity levels cross domain borders in a deterministic real-time manner. Thus it requires by its nature a decentralized engineering process. In order to manage the complexity, the correctness by construction approach is ensured with the fine specified ISS engineering steps. For a better cooperation between the development partners towards higher system dependability, approach of virtual front-loading is suggested. The engineering steps, virtual integration of FFA-model, derivation of FDA-model with the mapping procedure, prototyping on different platforms are supplemented with a systematic test methodic with consistent test cases through out the engineering process. The dependability E/E-architecture framework, common simulation, development and prototyping environment provide one of the most essential fundaments for the safety justified engineering process between the involved partners.

Hardware

The hardware architecture framework is addressed with the three aspects system topology, communication systems and ECU architecture. We tried to give a few vertexes and Lego-modules to provide a framework with possible configurations to dependable hardware architecture. A few use-cases are listed here to demonstrate the application of the hardware architecture framework.

The traditional work distribution is that OEM defines the system topology, communication systems and its interfaces to ECU, while the suppliers define and implement the ECU HW-architecture. With ISS, however, this distribution border is more and more unclear between the participants. Triggered by the aimed safety integrity level as to the safety norms, taking the constraints as development complexity into consideration, the hardware architecture framework should be worked out and configured with a much intensified interaction and cooperation between the development partners.

Software

The dependability software platform was specified in a layered reference model after a short benchmarking of automotive software with the evolution of IT. In order to design a hardware and application transparent fault tolerance, dependability software services such as ISS-communication service with redundancy, end-to-end CRC, Agreement Protocol, time partitioning and space partitioning for the integration of applications, fault management, fault treatment of dynamic reconfiguration and gateway services are introduced as an integrated manner and merged in the dependability software platform as mentioned above. The explanation about interaction and configuration of the dependability software services as a library concluded the software architecture framework.

Implication on the business model

The reaction of technologies and business process is always in mutual directions. The way how OEM and supplier should interact in the development of ISS must change in response to the E/E-architecture framework defined here. Again a benchmarking with the development of IT-industry shows a decentralized network in the development of ISS-application, with OEMs, suppliers and the third-party companies (e.g. software vendors and companies specialized for integration and test) can grant a more effective cooperation towards system with higher dependability. It brings also a few chances and challenges to the business model as well:

- Decentralized structure requires more effort to manage
- New market opportunities for suppliers esp. software supplier and system integrator
- Responsibility and warranty by the integration of safety components from different partners
- Systems integration and test can be offered as an independent service

11.2 Outlook for the future work

The concepts discussed in this dissertation provide potential for the further research, esp. taking the fact into the consideration that automotive standards and architecture usually takes a long time to establish, e.g. it took CAN 10 years to be introduced into the market and another 10 years to be widely used. The dependable E/E-architecture and engineering process will take a considerable time and adaptations to find their way in the real life.

The presented work in this thesis raises a few interesting topics for future research activities.

- Setting up a library of dependability services for the ISS-application from practice
- Further quantitative analysis of real-time performance of the dependability software services
- As mentioned in the Subsection 10.6, based on the experiences and finding during the validation, ISS Engineering Process and tool chain should be extended from the rapid prototyping to the later product phase of the serial development. One interesting topic is the safety analysis according to the special requirements of ISS.
- Further development of dynamic reconfiguration in the direction of service displacement
 - Concept of dynamic reconfiguration of communication paths for service reallocation
 - Resource reconfiguration, management and optimization for the service displacement

In all, the dependable E/E-architecture framework combined with the ISS Engineering Process is a promising approach for the development of future Integrated Safety Systems. Therefore, it has to find the way into real-world platform since an architecture framework provides only a reference model and guidelines for the practical solution. A step towards bringing the results closer to being deployed in products will be to advocate the use of the results as input and inspiration for on-going projects concerning automotive software platforms, such as the safety group in AUTOSAR [AUT07].

Appendix 1: Implementation details of the validator

In this chapter more details about the prototyping and validation of the safety concepts in this dissertation will be given. In order to keep an overview of the work in the scope the chapter, more details as to software implementation, simulation modes can be found the appropriate references and EASIS deliverables.

In Figure AI-1, the EASIS validator presented at the end of 2006 on the conference “Intelligent Transport System” in London with the most important nodes are shown.

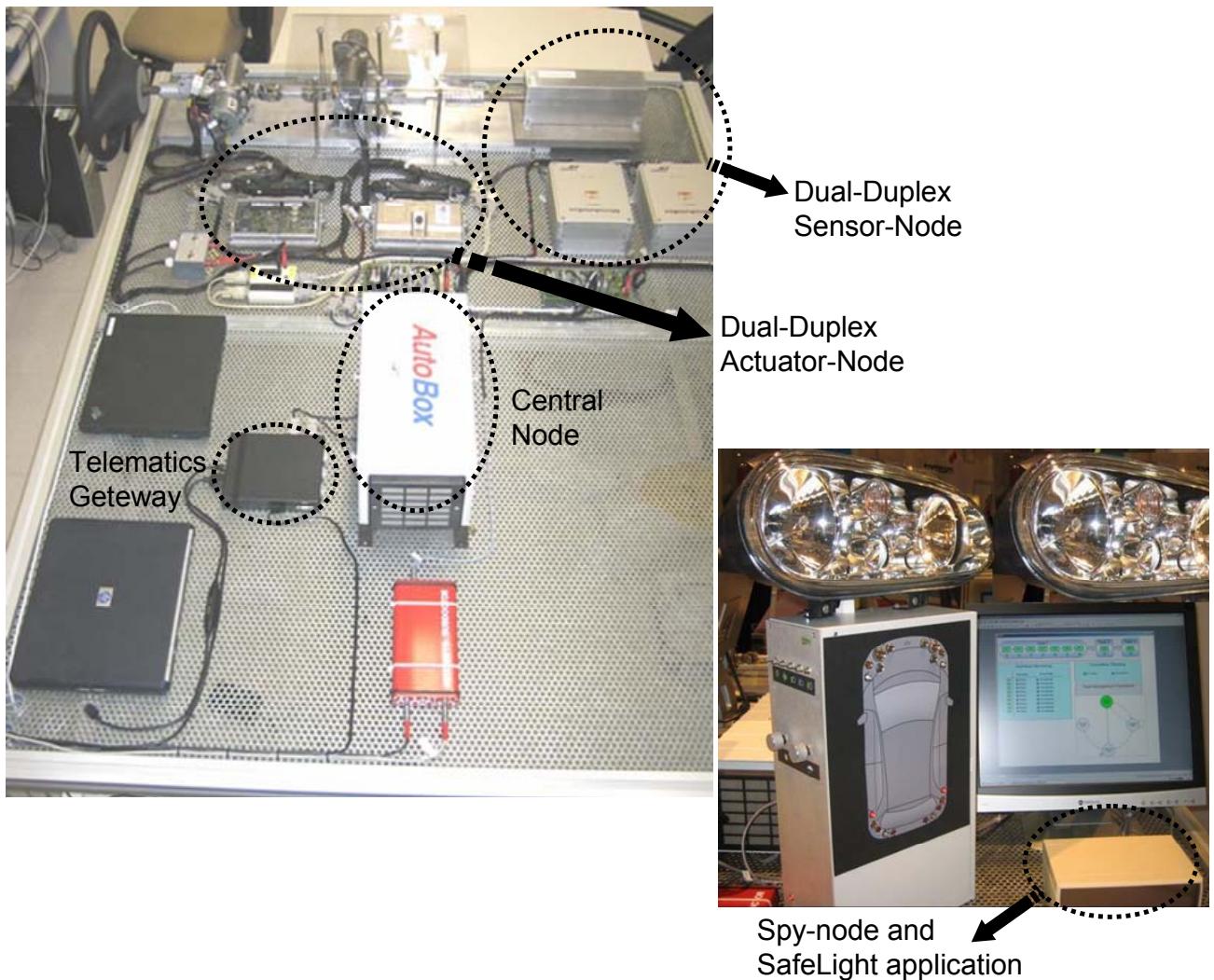


Figure AI-1: EASIS validator demonstrated on ITS 2006 in London

Appendix 1.1 Validation of hardware architectures for ISS

The dual-duplex 1oo2D ECU hardware architecture depicted in Subsection 8.3.1 Figure 8-8 is made up of 2 FSU. The hardware architecture of each FSU is demonstrated in Figure AI-2.

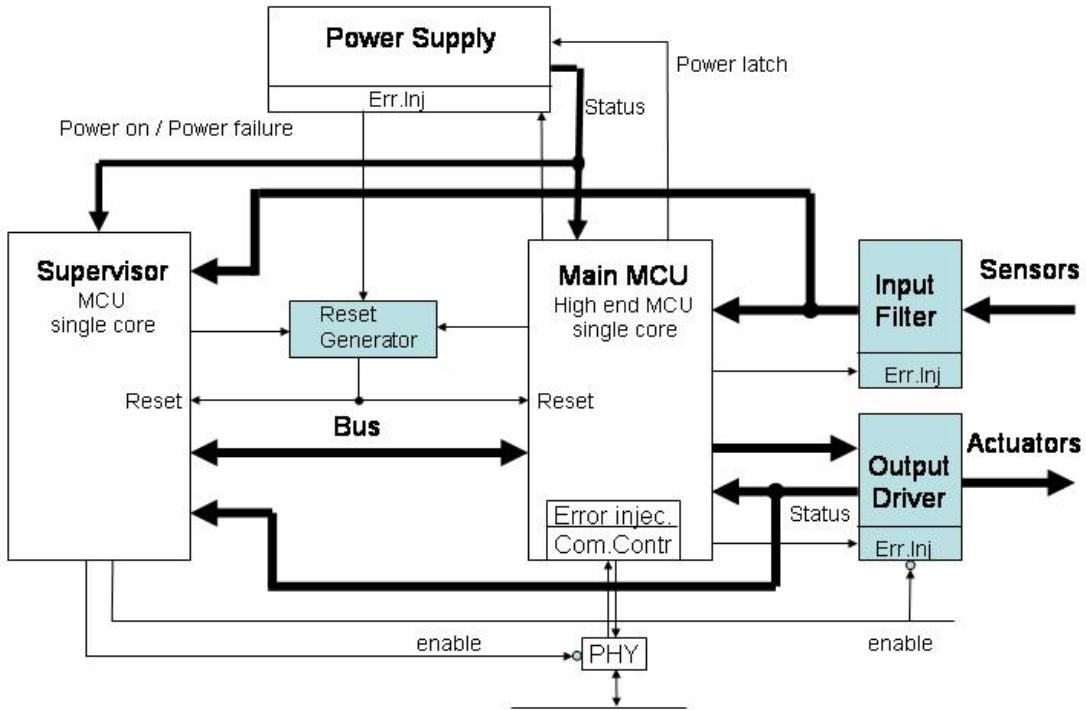


Figure AI-2: FSU processor block diagram [HWV06]

This power supply of FSU has two independent battery inputs (V_{b1} and V_{b2}) and generates the different supply voltages for the FSU internal circuitry. This block also generates the reset signals, which activates the logic devices once their supply voltages are stable, after the FSU has been switched on. It also generates the sensor power supply, both for the synchronous motor Hall Effect sensors and for the torque sensors. Feed-back on the status of these external supply lines is provided to the logic core of the FSU for fault handling. Finally, this block also provides power to the output stage, deriving it from one of the battery connections; should a failure occur on one of the FSU power supply lines, a switch over to the other power supply can take place.

The FSU core is made up of two units, namely a microcontroller managing the actual application software, the interface to the rest of the system, whilst a second device, a digital signal processor, acts as a slave to the former device, managing the interface to the output stage. The crosscheck mechanism let each unit monitor the operation of the other and can make the FSU enter a fail-safe state in case of fault. In this way, the processing core architecture of the FSU reflects the architecture depicted in Figure 8-8 of Subsection 8.3.1.

The validation of dual-duplex hardware architecture was carried out also according to the principle of fault injection. In the following a list of fault injection is given:

Test number	Test case	Description	Expected results
1.	Fail-silent and fail-operational by internal FSU fault	An internal simulated fault is injected in one of the two FSUs that belong to the fail-operational unit. The fault can be injected through the activation of an internal switch.	<u>Fail-silent characteristics:</u> The faulty FSU detects the fault, sends a diagnostic message on the FlexRay network and switches itself off. <u>Fail-operational characteristics:</u> The second FSU receives the diagnostic message, detects the fault of the other ECU and keeps the system fully operative.
2.	Fail-silent and fail-operational by communication fault	Both FlexRay channels of one fail silent node are disconnected.	<u>Fail-silent characteristics:</u> The faulty FSU detects the loss of communication with all the validator nodes and switches itself off. <u>Fail-operational characteristics:</u> The second FSU does not receive any diagnostic message, but detects the communication loss with the other FSU and keeps the system fully operative.
3.	Redundant communication with FlexRay	One of the two FlexRay channels of both fail silent nodes is disconnected.	The two fail-silent nodes detect the line disconnection and send a diagnostic message. Both the FSUs are fully operative.
4.	Redundant power supply by automatic switch	The system is supplied by line A. A fault is injected by disconnecting line A on one of the two fail silent FSUs.	The fail-silent FSU detects the fault and sends a diagnostic message. The switch is triggered. The two fail-silent nodes are fully operative.

Table AI-1: Fault injection for the validation of the HW-Architecture

Appendix 1.2 Validation of dependability software platform and services

As discussed in Subsection 10.3, the validation of dependability software platform and services are carried out based on the fault/error injection in the ISS-Applications SafeLane, SafeSpeed, SafeLight and the prototyping environment with a mixture of three fault injection methods as introduced in Subsection 10.3 [TRV06].

Appendix 1.2.1 Prototyping and validation of Agreement Protocol

As discussed in Subsection 10.3.1, the validation of Agreement Protocol is based on the coordinated state reconfiguration on the Central Node and the two Steering Wheel Sensor Nodes. The validation is the following three scenarios:

- All nodes agrees on consistent state information
- One node doesn't send any information / sends inconsistent information. The Agreement Protocol has to guarantee the toleration of one fault.
- More than one node doesn't send any information / sends inconsistent information. The Agreement Protocol helps to switch to a safe state by delivering status feedback of the agreement process.

The list of mechanisms of the Agreement Protocol service in the validator is given in the following as sensor and actuator control.

Test number	Test case	Description	Expected results
1.	Single Node Fault	The Central Node fails to receive one sensor signal	System detects single faulty sensor, fully operative
2.	Double Node Fault	The Central Node fails to receive both sensor signals	System detects faulty sensor input, fail-safe mode
3.	Single Byzantine Fault	One Sensor Node sends inconsistent information	System detects faulty sensor, fully operative
4.	Double Byzantine Fault	Two Sensor Nodes send inconsistent information	System detects faulty sensor, fail-safe mode

Table AI-2: Test cases for the validation of Agreement Protocol

In the test cases 1 and 2, the communication between Central Node and Sensor Nodes are carried out with Agreement Protocol, but they mainly test the system wide coordinated reconfiguration. In the following, test cases 3 and 4, specially designed for Byzantine Fault are specified in details.

Test Case 3 – Single Byzantine Fault

Figure AI-3 shows the System State when a Byzantine fault is injected in one Steering Wheel Sensor node (the node sends different information to the other Steering Wheel Sensor node and to the Central Node). The node tries to reinitialize (five attempts) and then fixes its state on faulty. Central Node recognizes the situation, sets the flag of “Faulty Sensor Node” but since the other Steering Wheel Sensor node is still working, complete steering function and SafeLane functions are working.

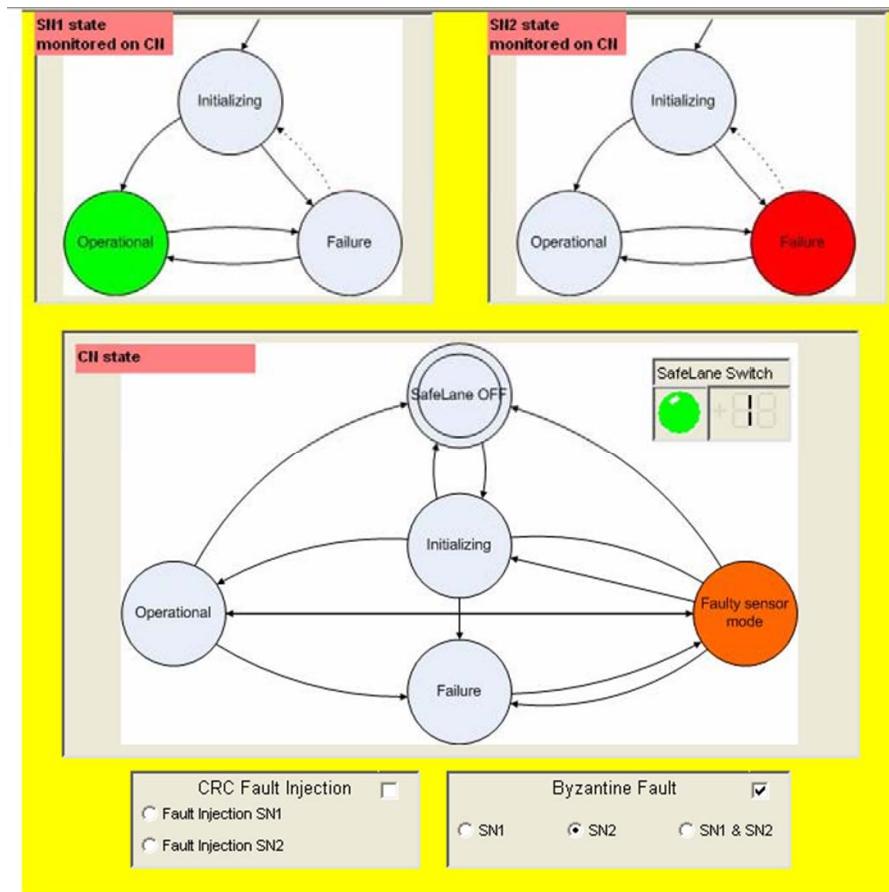


Figure AI-3: Single Byzantine Fault

Test Case 4 – Double Byzantine Fault

Figure AI-4 demonstrates the System State when Byzantine faults are injected in both Steering Wheel Sensor nodes (both nodes send different information to the other Steering Wheel Sensor node and to the Central Node). Both Steering Wheel Sensor nodes try to reinitialize (five attempts) and then fix their states on faulty. Central Node recognizes the situation, sets the flag of “Faulty”. Therefore, SafeLane functions are disabled and steering function goes to the fail-safe mode.

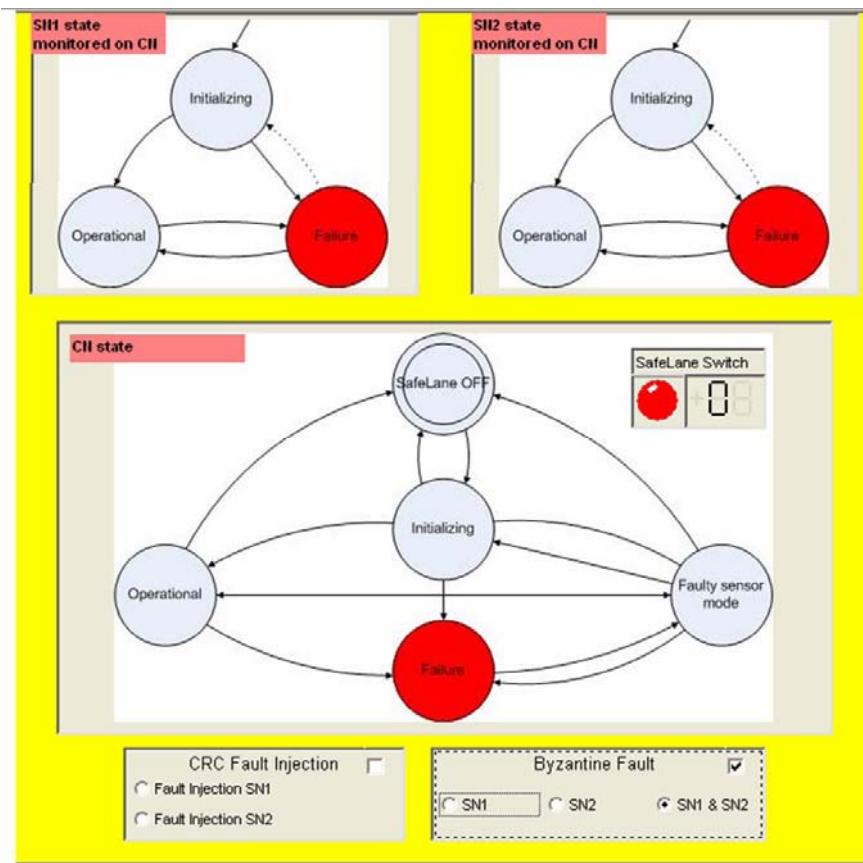


Figure AI-4: Double Byzantine Fault

Appendix 1.2.2 Prototyping and validation of Software Watchdog

As specified in Subsection 10.3.2, the SW-Watchdog has been tested on the Spy-Node in the context of a simplified SafeLight application. The SW-Watchdog is integrated with the Fault Management Framework. Complete test results on FMF validation are detailed in Appendix 1.2.3. The SW-Watchdog is configured to monitor all the Runnables: (Aliveness Monitoring, Arrival Rate Monitoring and Control Flow Checking).

The demonstrative application is composed of three tasks that manage nine runnables:

- Task T1 (execution period = 0200 ms) for Runnables R1, R2, R3, R4, R5, R6, R7
- Task T2 (execution period = 1000 ms) for Runnable R8
- Task T3 (execution period = 2000 ms) for Runnable R9

For each runnable of the demonstration application, three errors are defined: Aliveness Error, Arrival Rate Error and Control Flow Error (e.g. Runnable R1 is associated with R1_Aliveness, R1_Arrival_Rate and R1_Control_Flow errors). Using the HMI of the Control PC we can:

- Display in real time the current internal values of the heartbeat monitoring counters (AC, CCA, ARC, CCAR) and the configured parameters (m, n, p, q)
- Display the last executed runnable of each task

- Display in real time the current internal state of each error handled by the Fault State Manager associated with the SW Watchdog
- Inject a set of predefined faults : deactivation of runnables, inversion of order of execution of runnables, modification of the execution period of runnables
- Execute the scheduler cycle by cycle in order to see the evolution of the counters step by step

Test Case 1 – Aliveness Monitoring

The SW-Watchdog monitors the aliveness of runnables. The execution period of the SW-Watchdog defines the monitoring cycle. If the number of executions of a runnable (counted by the Aliveness Counter AC) is lower than the configured limit (m), which is called by this runnable within a certain monitoring period, an error will be detected. The monitoring period is defined in number of monitoring cycles (counted with the Cycle Counter for Aliveness CCA).

In the configuration of test environment, the execution period of Runnable R8, which belongs to Task T2, was set to 1000ms. The monitoring cycle is 100ms. The fault assumption is “less than 10 executions of R8 during 10,000 ms”. So the minimum number of executions m is 10 and the monitoring period for R8 is $10,000 \text{ ms} = n * \text{monitoring cycle} = 100 * 100 \text{ ms}$. As depicted in Figure AI-5, for the Aliveness checking of R8, the number of monitoring cycles n is chosen as 100.

The screenshot shows a software interface titled "Software Watchdog". It contains three main tables: HBM, Aliveness, and Arrival Rate.

HBM		Aliveness				Arrival Rate			
Runnable	Task	AC	m	CCA	n	ARC	p	CCAR	q
R1	1	7	10	15	20	2	5	5	10
R2	1	7	10	15	20	2	5	5	10
R3	1	7	10	15	20	2	5	5	10
R4	1	7	10	15	20	2	5	5	10
R5	1	7	10	15	20	2	5	5	10
R6	1	7	10	15	20	2	5	5	10
R7	1	7	10	15	20	2	5	5	10
R8	2	9	10	95	100	4	5	45	50
R9	3	4	5	95	100	4	5	95	100

CFC	
Task	Last Runnable
T1	R7
T2	R8
T3	R9

Figure AI-5: Aliveness Monitoring without error

The fault Period $T_2 * 2$ is injected. Now the execution period of R8 is 2000 ms instead of 1000 ms, so R8 will be executed two times less than expected. After the monitoring period of 10,000 ms, R8 has been executed only 5 times (less than the low limit $m = 10$) and the error R8_Aliveness is detected.

Test Case 2 – Arrival Rate Monitoring

The SW-Watchdog monitors the arrival rate of runnables. If the number of executions of a runnable (counted by the Arrival Rate Counter ARC) is higher than the configured limit (p), an error is detected. The number of monitoring cycles is registered with the Cycle Counter for Arrival Rate CCAR. Here, the monitoring period equals the monitoring cycle, which means that ARC is checked at each monitoring cycle.

In the configuration of test environment, the execution period of runnable R8 in task T2 is 1000 ms, while the monitoring cycle is 100 ms. The fault assumption is “more than 5 executions of R8 during 5000 ms”. So the maximum number of executions p is 5 during 5000 ms = $q \cdot \text{monitoring cycle} = 50 \cdot 100 \text{ ms}$, so q is 50.

By the fault injection period $T_2 / 2$ is set. Now the execution period of R8 is 500 ms instead of 1000 ms, thus R8 will be executed twice as often as expected. As the arrival rate checking is performed at each monitoring cycle, after 2600 ms R8 has been executed 5 times and the error R8_Arrival_Rate is detected.

Test Case 3 – Control Flow Checking

The SW-Watchdog with the function Control Flow Checking monitors the execution sequence of the runnables inside a task. For task T1 the expected execution order is R1 R2 R3 R4 R5 R6 R7 R1 R2 R3 etc.

- Fault injection test case 1

The fault *R3 disabled* is injected; the following errors are detected by the SW-Watchdog:

- R4_Control_Flow (because R4 is not the successor of R2): after the monitoring period for R3 (where the Aliveness checking is performed for R3), another error is detected
- R3_Aliveness (because R3 has not been executed)

- Fault injection test case 2

The fault “inverse R1 / R2” is injected; the following errors are detected by the SW-Watchdog:

- R1_Control_Flow (because R1 is not the successor of R2)
- R2_Control_Flow (because R2 is not the successor of R7)
- R3_Control_Flow (because R3 is not the successor of R1)

Appendix 1.2.3 Prototyping and validation of Fault Management Framework

The Fault Management Framework is tested on the Spy-Node in the prototyping environment as for the SW-Watchdog. The test cases are organized in three scenarios.

1. Test of the Fault State Manager part
2. Test of the Supervision – Reconfiguration part

A synthesis test demonstrates an integration example of the Fault Management Framework with the demonstration application:

3. Test of the Fault Management Framework

The list of characteristics of the Fault Management Framework tested in the validator is given in the following tables.

Test number	Test Case	Description	Expected results
1	Off line configuration	Demonstration of the Python tool to generate configuration source files for Fault State Manager software component	Functional code generated
2	Filtering mechanism and error state management	Demonstration of the filter counter, demonstration of “No Error, Present, Present & Confirmed, and Confirmed states management”	Functional filter counter
3	Combined errors	Demonstration of resilience to logical errors combination	Functional code generated

Table AI-3: Test cases for the Fault State Manager in FMF

Test number	Test Case	Description	Expected results
4	Mode management by dynamic reconfiguration of fail-degradation	Demonstration of call-back function mechanism for supervision integration, demonstration of degraded mode	Functional call back mechanism working
5	Dynamic reconfiguration on the task layer	Demonstration of disabling tasks and scheduler reconfiguration, demonstration of SW-Watchdog reconfiguration	Functional reconfigurable scheduler working
6	Dynamic reconfiguration with ECU reset	Demonstration of reset capability	Reset capability with context-awareness

Table AI-4: Test cases Supervision – Reconfiguration in FMF

In the following the test cases 4, 5 and 6 will be specified in details:

Test Case 4 – Mode management by dynamic reconfiguration of fail-degradation

By association an error with a call back function, the Fault State Manager calls this function when the error passes to the state “Confirmed”.

```
conf.SetEventCallBack( 'SWCIOIN_ERR_STS_LAMP_INDIC_FL', ('SUP_vidErrorStsLampIndicFL',) )
conf.SetEventCallBack( 'SWCIOIN_ERR_STS_LAMP_SIDE_FL', ('SUP_vidErrorStsLampSideFL',) )
conf.SetEventCallBack( 'SWCIOIN_ERR_STS_LAMP_LBEAM_FL', ('SUP_vidErrorStsLampLBeamFL',) )
conf.SetEventCallBack( 'SWCIOIN_ERR_STS_LAMP_HBEAM_FL', ('SUP_vidErrorStsLampHBeamFL',) )
conf.SetEventCallBack( 'SWCIOIN_ERR_STS_LAMP_FOG_FL', ('SUP_vidErrorStsLampFogFL',) )
```

This design mechanism is used to integrate the Fault State Manager and the Supervision and Reconfiguration sub-components. The management of the demonstration application modes (Normal mode, Degraded mode, Restarted mode, Fail safe mode) is realized in the Supervision sub-component. For example as depicted in Figure AI-6, when the error STS_LAM_INDIC_FL (open load detection on front left indicator lamp) is detected, the application passes in a ‘Degraded mode’ (front left position lamp replaces front left indicator lamp for left turning indication).

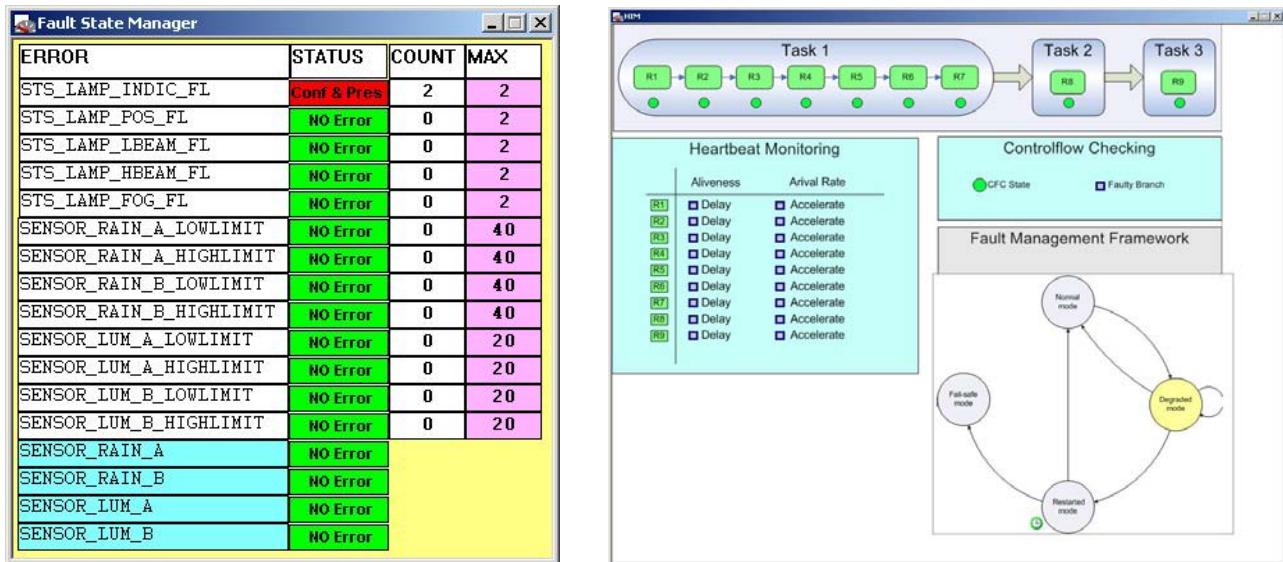


Figure AI-6: Supervision Reconfiguration – Mode management

Test Case 5 – Dynamic reconfiguration on the task layer

In this SafeLight application, runnables correspond to elementary atomic treatments that are executed periodically (e.g. Runnable R1 reads inputs, Runnable R2 manages blinking, ...). Runnables of the same period of the same safety integrity are grouped in a task (e.g. Task T1 200ms, Task T2 1000 ms, etc.) The tasks are organized functionally in configurations that correspond to a set of tasks executed in a major stable state (e.g. Normal Configuration, Fail Safe Configuration, etc.). A synchronous real-time scheduler manages the execution of all tasks of a configuration with an elementary cycle of 100 ms. The current configuration can be switched. A task of a configuration can be activated / deactivated.

Two types of reconfiguration of the execution flow have been tested:

- Deactivation of tasks. The tasks associated with the automatic lighting functions are deactivated in case of sensor fault detection. As depicted in Figure AI-7, the deactivation of task1 leads to the state transition of SafeLight to the fail-safe mode.
- Switch of configuration. The application switches in a “Fail Safe Configuration” in case of non recoverable software fault.

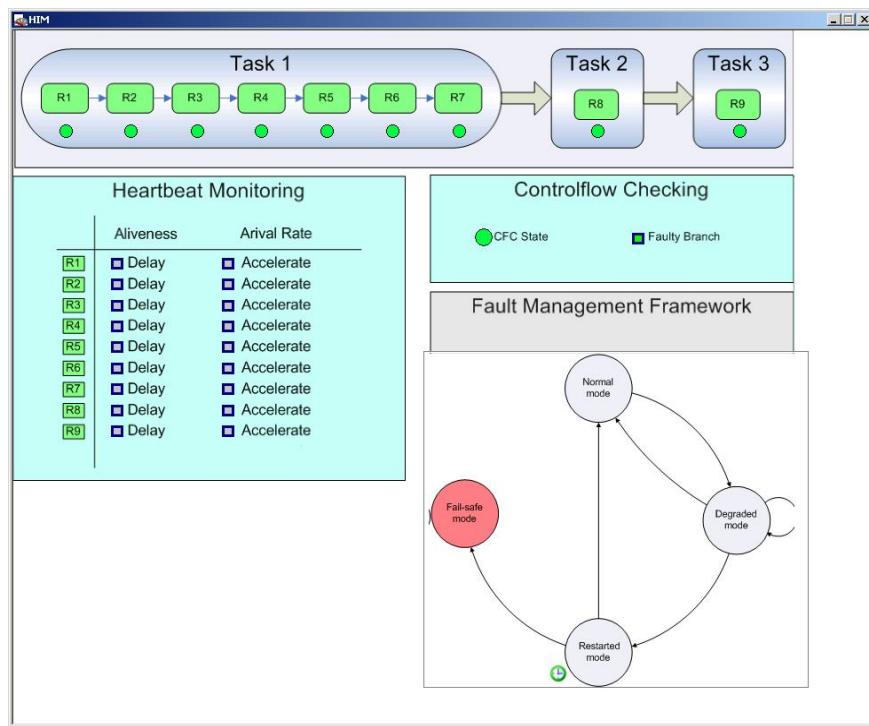


Figure AI-7: Supervision Reconfiguration – Scheduler

Test Case 5 – Dynamic reconfiguration with ECU reset

The supervision and reconfiguration unit in FMF can trigger an ECU reset. In this implementation, the design solution was to use the internal hardware watchdog of the micro-controller of the ECU. In the demonstration application, the ECU reset is used as a recovering strategy for some critical software failures, e.g. a critical control flow error as depicted in Figure AI-8.

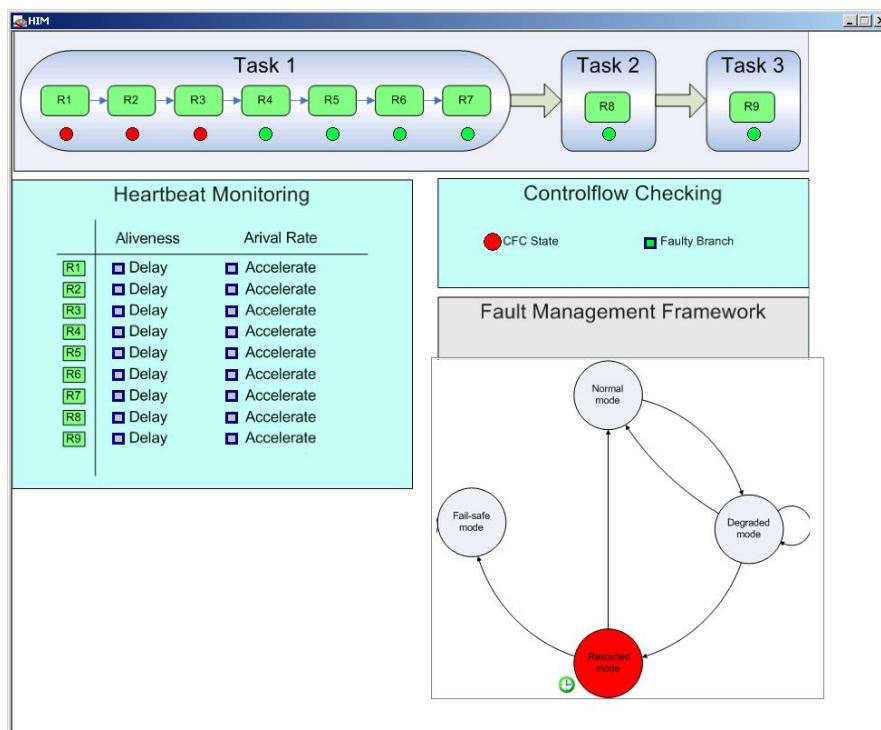


Figure AI-8: Supervision Reconfiguration – ECU Reset

Appendix 2: Mathematic derivation of the communication overhead of agreement protocol

$$\begin{aligned}
 & \underbrace{n \cdot (n-1)}_{\text{1st Round}} + \underbrace{n \cdot (n-1)^2}_{\text{2nd Round}} + \dots + \underbrace{n \cdot (n-1)^{m+1}}_{(m+1)\text{-th Round}} = \\
 & n \cdot ((n-1) + (n-1)^2 + \dots + (n-1)^{m+1}) = \\
 & \frac{n \cdot (n-1)((n-1)^{m+1} - 1)}{n-2}
 \end{aligned}$$

Figure All-1: Derivation of the communication overhead for the OM

$$\begin{aligned}
 & \underbrace{n \cdot (n-1)}_{\text{1st Round}} + \underbrace{n \cdot (n-1) \cdot (n-2)}_{\text{2nd Round}} + \dots + \underbrace{n \cdot (n-1) \cdot \dots \cdot (n-m-1)}_{(m+1)\text{-th Round}} = \\
 & \frac{n!}{(n-2)!} + \frac{n!}{(n-3)!} + \dots + \frac{n!}{(n-m-2)!}
 \end{aligned}$$

Figure All-2: Derivation of the communication overhead for the SM

Appendix 3: Glossary

A few terms used in this dissertation are explained here as a short summary. For more complete and detailed glossary please refer to EASIS glossary [Glo07] and AUTOSAR glossary [Glo03].

Application Software Component

An Application Software Component is a specific Software Component realizing a defined functionality of a set (one or more) of features, which consist out of 1..n ($n \geq 1$) functions (in terms of C-Code functions) and 1..m ($m \geq 1$) executable runnables. In EASIS, Application Software Components are resident in the most upper layer of EASIS software topology.

Basic Software Module

Basic Software Module provides the infrastructural (schematic dependent and schematic independent) functionalities of an ECU. It consists of ECU Firmware and Standard Software.

Compositionality

Compositionality is given when the behavior of a software component or subsystem of a system has to be independent of the overall system load and configuration. Notes: Compositionality is an important property of deterministic systems. This property leads to a complete decoupling of systems. Smooth subsystem integration without backlashes is then easily achievable.

Software Configuration

The configuration of software elements in a software system. Notes: A software element is a clearly definable software part. A software configuration is a selected version of software modules, software components, parameters and generator configurations. Calibration and variant coding can be regarded as subset of software configuration.

Control Flow

The directed transmission of information between multiple entities, directly resulting in a state change of the receiving entity. Notes: A state change could result in an activation of a schedulable entity.

Event

The occurrence of a state change of a hardware or software entity or system or its environment. Notes: (1) Examples of different types of events: Failure, exception, interrupt, hazardous event mishap, loss event (2) Event is a specialization of Signal.

Fail-Degraded

Property of a system or functional unit, describes the ability of a system to continue with intended degraded operation at its output interfaces despite the presence of hardware or software faults. Notes: e.g. "Limp home" functionality for ECU (reduce torque to assure an arrival at home or service station).

Fail-Operational

Property of a system or functional unit, describes the ability of a system or functional unit to continue normal operation at its output interfaces despite the presence of hardware or software faults. This is a characteristic of a unit that means that the system is designed in such a way, that it can (without repair) fail once and remain fully operational without any limit of time, e.g. braking system

Fail-Safe

Property of a system or functional unit. In case of a fault the system or functional unit transits to a safe state. This is the behavior of a unit such that the outputs of the unit must be correct or otherwise must be in any electrical or logical state that is supposed to lead the vehicle in an intrinsic safe state.

Fail-Silent

Property of a system or functional unit. In case of a fault the output interfaces are disabled in a way that no further outputs are made. This is the behavior of a unit such that the outputs of the unit must be correct or otherwise must be "silent" that is null or in any other electrical or logical state that does not lead to any kind of actuation or interference with other parts of the system. This applies either to the communication bus or to sensors and actuators directly connected to the unit. This is a special case of fail safe. Fail-silent is a special case of the fail-safe property.

Fault Containment

As an arbitrary propagation of errors cannot be tolerated, fault containment regions, which restrict errors to certain parts of the system, are needed. Methods for error containment are access-restriction, consistency checking, physical isolation, and others.

Front Loading

During a standard product development process development and validation/verification workload is steadily increasing over time until it reaches its maximum shortly before start-of-production. This may lead to "fire-fighting" in some projects, where unexpected trouble occurs during validation. To solve this issue, a maximum of validation and verification should be done in early phases of the development process as early as possible

Redundancy

Existence of means, in addition to the means which would be sufficient, for a system or functional unit to perform a required function.

Appendix 4: List of figures

Figure 1-1: Development of active, passive and integrated safety electronic systems.....	1
Figure 1-2: Targets set by EU of 50% reduction in the road fatalities before 2010	2
Figure 1-3: Structure of the focus in the dissertation	5
Figure 2-1: The dependability tree [Lap95]	7
Figure 2-2: Difference between fault, error and failure	8
Figure 2-3: Timing requirement of fault tolerance	9
Figure 2-4: Relationship between different operational modes of an ECU	10
Figure 2-5: Layered architecture model of electronics	13
Figure 3-1: ECU topology of Mercedes E-class BR211	21
Figure 3-2: Hardware components of an ECU.....	22
Figure 3-3: Single-Processor ECU.....	22
Figure 3-4: Dual-Processor ECU	23
Figure 3-5: Triple-Processor ECU as a full 2oo3-system.....	24
Figure 3-6: Triple-Processor ECU.....	25
Figure 3-7: Two variant of Dual-Duplex-Processor ECU	25
Figure 3-8: FlexRay bus access and communication cycle	29
Figure 3-9: FlexRay topologies	30
Figure 3-10: Software topology for Mercedes-Benz cabin ECU [Ruh04].....	33
Figure 3-11: EAST architecture framework.....	34
Figure 3-12: AUTOSAR architecture framework [SWA06].....	35
Figure 3-13: The V-Model	39
Figure 3-14: Model based development process [Ruh04]	39
Figure 3-15: Basic V-style process model integrated in lifecycle engineering process	41
Figure 3-16: Development process according to [Hed01].....	41
Figure 3-17: "Double-V-Model" according to [Ben04]	42
Figure 3-18: Calculation of Safe Failure Fraction according to IEC61508.....	45
Figure 3-19: Overall framework of the safety lifecycle in ISO26262 [IEC01]	47
Figure 3-20: Reference phase model for the development of a safety-related item	48
Figure 3-21: Preliminary mapping of safety scales between IEC 61508 and ISO 26262	49
Figure 5-1: Framework of dependability activities.....	54
Figure 5-2: V-model indicating inner and outer loop	54
Figure 6-1: Simplified fault hypothesis of a standard ECU	58

Figure 6-2: Cause of Byzantine fault with “1/2” area threshold.....	61
Figure 6-3: Gate transfer function with “1/2” area	61
Figure 6-4: Byzantine Faults in Schrödinger’s CRC [SIV04].....	62
Figure 7-1: Overview of ISS Engineering Process with virtual front-loading.....	65
Figure 7-2: ISS Engineering Process with virtual front-loading in details	67
Figure 7-3: Notation in the ISS Engineering Process	69
Figure 7-4: Global view of ISS Engineering Process	70
Figure 7-5: Specify preliminary requirements	71
Figure 7-6: Safety integrated requirement specification and development of the FAA-model.....	73
Figure 7-7: Mapping of system-PFH to FAA-components	76
Figure 7-8: Development steps for hardware architecture	78
Figure 7-9: Design and validation of Functional Architecture Model with virtual front-loading.....	80
Figure 7-10: Refinement of FDA model with dependable SW-platform	84
Figure 7-11: Process step 5.1 under magnifying glass.....	85
Figure 7-12: Validation of FDA-model with software-in-the-loop (SiL) test.....	86
Figure 7-13: Test in different stages	87
Figure 7-14: Validation of FDA-model with hardware-in-the-loop test	89
Figure 7-15: OEM - Supplier Workflow	102
Figure 7-16: OEM view in the design of electric/electronic architecture	103
Figure 8-1: Backbone architecture.....	107
Figure 8-2: Central gateway architecture.....	107
Figure 8-3: Multi gateway architecture.....	108
Figure 8-4: Different sampling of two sensors	111
Figure 8-5: Variant of Dual-Duplex-ECU as 2oo2D-system.....	114
Figure 8-6: 2oo2D-system with diversity in hardware	114
Figure 8-7: Variant of Dual-Duplex-ECU as 1oo2D-system.....	115
Figure 8-8: Implementation of Dual-Duplex-ECU as 1oo2D-system [HWAA06].....	116
Figure 8-9: Example of the actuator monitoring (electronic motor).....	118
Figure 8-10: Examples of ISS application and mapping to the vehicle domains	120
Figure 8-11: Distribution of SafeSpeed to the HW-architecture (FAA → FDA).....	122
Figure 8-12: Exemplary system topology of future mid size car [HWAA06].....	122
Figure 8-13: Communication and functional architecture of SafeSteering.....	123
Figure 8-14: Exemplary system topology of future large size car [HWAA06]	124
Figure 8-15: Exemplary 1oo2D dual-core ECU architecture.....	125

Figure 9-1: EASIS layered software topology [SWP06]	128
Figure 9-2: EASIS software topology –software components and services [SWT06]	129
Figure 9-3: Mapping App.SW-Cs of different ASILs to the hardware topology	131
Figure 9-4: Example scenario of intra-ECU and inter-ECU communication	132
Figure 9-5: Communication redundancy mechanisms.....	133
Figure 9-6: Reference model for end-to-end communication checksum	134
Figure 9-7: Data flow from application SW-C to transceiver (sender).....	135
Figure 9-8: Data flow from transceiver to application SW-C (receiver).....	135
Figure 9-9: Date flow with application CRC on the sender	136
Figure 9-10: Fault masking 2-out-of-3 system [Ech90]	139
Figure 9-11: OM algorithm with $n = 4$ and $m = 1$	141
Figure 9-12: Time diagram of the Signed Message Protocol with 3 nodes	142
Figure 9-13: Pendulum Protocol for 3 nodes with $\delta = 2$	143
Figure 9-14: Interface to fault-tolerant communication and application	145
Figure 9-15: Assembly symbol in UML with Enterprise Architect	145
Figure 9-16: Software partitioning in the dependable software architecture.....	147
Figure 9-17: Mapping of application to ECUs and runnables to tasks	148
Figure 9-18: UML-model of objects in operating system	149
Figure 9-19: Memory protection of indirect data access	150
Figure 9-20: Definition of task execution time	151
Figure 9-21: Control flow check of runnables	153
Figure 9-22: Different units constituting the SW-Watchdog service.....	154
Figure 9-23: Activity diagram for Aliveness Monitoring	156
Figure 9-24: Activity diagram for Arrival Rate Monitoring	157
Figure 9-25: Example of Heartbeat Monitoring	158
Figure 9-26: Sequence diagram for Control Flow Checking	160
Figure 9-27: Sequence diagram for TSIU	161
Figure 9-28: Fault Management Framework Structure	163
Figure 9-29: FSM structure and ID mapping.....	164
Figure 9-30: Event dispatcher example	165
Figure 9-31: HW/SW mapping of the FMF	166
Figure 9-32: Overview of supervision	167
Figure 9-33: Mapping of Errors to Configurations	171
Figure 9-34: Interface of basic, dependability and gateway software services.....	176

Figure 10-1: Tool chain and development process.....	179
Figure 10-2: System topology of EASIS architecture validator [AfS07]	181
Figure 10-3: Fail-safe unit of dual-duplex actuator ECU.....	183
Figure 10-4: Mapping between faults and errors	185
Figure 10-5: Outline of system architecture for the validation of Agreement Protocol Service.....	186
Figure 10-6: User interface for agreement protocol validation.....	186
Figure 10-7: Modeling of runnables and program flow in SafeSpeed	187
Figure 10-8: Test with injected aliveness error	188
Figure 10-9: Collaboration of fault detection units	189
Figure 10-10: Evaluation cases with fault injection for FMF	190
Figure 10-11: Fault injection triggered with CANoe to FMF	191
Figure 10-12: FAA-model of SafeSpeed	192
Figure 10-13: FDA-model of the SpeedControl in SafeSpeed	193
Figure 10-14: FlexRay communication scheduling in the validator.....	194
Figure AI-1: EASIS validator demonstrated on ITS 2006 in London.....	200
Figure AI-2: FSU processor block diagram [HWV06]	201
Figure AI-3: Single Byzantine Fault	204
Figure AI-4: Double Byzantine Fault	205
Figure AI-5: Aliveness Monitoring without error	206
Figure AI-6: Supervision Reconfiguration – Mode management	209
Figure AI-7: Supervision Reconfiguration – Scheduler	210
Figure AI-8: Supervision Reconfiguration – ECU Reset	210
Figure AII-1: Derivation of the communication overhead for the OM	211
Figure AII-2: Derivation of the communication overhead for the SM	211

Appendix 5: Literature index

- [EUC01] EU, "White Paper - European transport policy for 2010, time to decide", European Commission, Office for Official Publications of the European Communities, Luxemburg, 2001, <http://europa.eu.int>
- [ADS07] M. Lannoije, J. Dr. Schuller, et. al., "Entwurf und Realisierung des Funktions- und Sicherheitskonzepts der Audi Dynamiklenkung", presented at Haus der Technik, 27. Conference "Elektronik im Kraftfahrzeug" (In-vehicle Electronics), Dresden, Germany, 2007
- [ADL05] W. Bernhardt, H. Erl, "Markt- und Technologiestudie Leistungselektronik Automotive 2015", Arthur D Little GmbH - Consulting, 26. Sep. 2005
- [AfS07] J. Stroop, X. Chen, et. al., "Architectures for Safety", January, 2007, http://www.easis-online.org/wEnglish/img/pdf-files/dSPACENEWS2007-1_Architectures_for_Safety_en_504
- [AOS06] AUTOSAR, "AUTOSAR Specification of Operating System", 2006 http://www.autosar.org/download/AUTOSAR_SWS_OS.pdf
- [ASM06] ASAM, "Association for Standardization of Automation and Measuring Systems", 2006, <http://www.asam.net>
- [ASR04] ASRB, Automotive Safety Restraints Bus, 2004, <http://www.electronic-data.com/unternehmen/4257.asp>
- [ATZ05] ATZ/MTZ, "Aktive und passive Sicherheit," ATZ/MTZ extra S-Klasse BR221, pp. 118-125, 2005
- [AUT07] AUTOSAR, "AUTOSAR - Automotive Open System Architecture", 2007, www.autosar.org
- [ATO07] AUTOSAR, "AUTOSAR – Technical Overview", 2007, http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf
- [Bel01] R. Belschner, et al., "FlexRay - The Communication System for Advanced Automotive Control Systems", presented at SAE Technical Paper Series Nr. 2001-01-0679, Detroit, 2001
- [Ben04] S. Benz, "Eine Entwicklungsmethodik für sicherheitsrelevante Elektroniksysteme im Automobil", Dissertation in Institut für Technik der Informationsverarbeitung: Universität Karlsruhe, 2004
- [BER04] J. Ferreira, "An Experiment to Assess Bit Error Rate in CAN", presented at RTN 2004 - 3rd Int. Workshop on Real-Time Networks, 2004
- [Bes02] A. Best, Echtle, K., "Standardverfahren und spezielle Ansätze zur Fehlertoleranz von Steuergeräten in Automobilen", Projekt FeSTA, Daimler AG, Uni Essen, 2002
- [Bro03] M. Broy, "Automotive Software and Systems Engineering", presented at 25th International Conference on Software Engineering, 2003, pp. 719 - 720

- [BSW06] EASIS, "Deliverable D1.2-2 - Basic services", EASIS Partnership, 2006
www.easis.org
- [Gri00] R. Grießbacher, "Byteflight - neues Datenbussystem für sicherheitsrelevante Anwendungen", Automotive Electronics of ATZ/MTZ, 01/2000,
http://www.byteflight.com/presentations/atz_sonderausgabe.pdf
- [CHS06] X. Chen, M. Limam, M. Wedel, et. al., "Concept and Integration of an Agreement Protocol in a Dependable Software Platform for Automotive Integrated Safety Systems", presented at Automotive - Safety & Security 2006, Stuttgart, Germany, 2006,
http://www.easis-online.org/wEnglish/img/pdf-files/Paper_Automotive-Safety&Security2006.pdf
- [CoW05] Freescale, "CodeWarrior™ Development Studio for Freescale™ HCS12(X) Microcontrollers, product information", 2005
- [Con06] Conti Temic microelectronic GmbH 2006, Electronic Brake Systems MK25,
http://www.conti-online.com/generator/www/de/en/cas/cas/themes/products/electronic_brake_and_safety_systems/electronic_brake_systems/abs_tcs_esc/ebs_1003_en.html
- [Cri91] Cristian, F. „Understanding Fault Tolerant Distributed Systems“, Communications of the ACM, Ausgabe 34, Heft 2, 1991, pp. 56-78
- [DFP06] EASIS, "Deliverable D1.2-13 - Description of fault types", EASIS Partnership, 2006, <http://www.easis.org>
- [DIN93] DIN60870-5-1, "Norm Fernwirkeinrichtungen und -Systeme Teil 5: Übertragungsprotokoll", 1993
- [DSC05] Decomsys, "Simcom User Manual Version 2.2, „DECOMSYS::SIMCOM<FlexRay> Matlab/Simulink blockset for FlexRay, Generator User Manual Version 2.2.8, Designer User Manual Version 2.0.6, Simsystem User Manual Version 2.2", Dependable Computer Systems GmbH, Vienna 2005
- [DyR96] D. B. Stewart, G. Arora, "Dynamically Reconfigurable Embedded Software - Does It Make Sense?" presented at IEEE Intl. Conf. on Engineering of Complex Computer Systems and Real Time Application Workshop, Montreal, Canada, 1996, pp. 217-220,
- [EAD42] EASIS Deliverable D4.2 A Prototype Tool Interaction Software Layer
- [EAD41] EASIS Deliverable D4.1 Appendix 1: EASIS Engineering Process
- [EAST03] EAST-EEA Embedded Electronic Architecture "Glossary", Version 6.1, ITEA EAST-EEA Project, www.east-eea.net, Version 7.2, 2003
- [Ech90] Echtle, K. „Fehlertoleranzverfahren“, Springer Verlag, Berlin [u.a.], 1990, ISBN 3-540-52680-3
- [Eck06] M. Eckmann, F. Mertens, "Close-to-Production Prototyping, Flexible and Cost-efficient", in ATZ Elektronik, vol. 01/2006, 2006, pp. 22-27

- [Ehr03] J. Ehret, "Validation of Safety-Critical Distributed Real-Time Systems", Dissertation in Electrical and Computer Engineering. Munich: Technische Universität München, 2003
- [EiK06] X. Chen, et. al., "A Dependable Software Platform for Automotive Integrated Safety Systems", presented at Haus der Technik, 26. Conference "Elektronik im Kraftfahrzeug" (In-vehicle Electronics), Dresden, Germany, 2006, http://www.easis-online.org/wEnglish/img/pdf-files/Paper_Elektronik_im_Fahrzeug.pdf
- [EiK07] X. Chen, F. Salewski, et. al., "Concept and Prototyping of a Fault Management Framework for Automotive Safety Relevant Systems", presented at Haus der Technik, 27. Conference "Elektronik im Kraftfahrzeug" (In-vehicle Electronics), Dresden, Germany, 2007, http://www.easis-online.org/wEnglish/img/pdf-files/Paper_Elektronik_im_Fahrzeug_2007.pdf
- [EIA04] Elektronik automotive, "Systemvernetzung: Künftig nur noch 20 Steuergeräte", 3/2004, vol. Heft 3, pp. 21, 2004
- [Ele04] B. Elend, "FlexRay Netzwerk Topologie für sicherheitsrelevante Applikationen", auto&elektronik, vol. 2/2004, pp. 44-46, 2004, http://www.flexray.com/publications/Philips_FR_auto_elektronik02_2004.pdf
- [FMF06] EASIS, "Deliverable D1.2-8 - Fault management framework", EASIS Partnership, 2006, http://www.easis-online.org/wEnglish/download/Deliverables/EASIS_Deliverable_D1.2-8_V1.0.pdf
- [FTD06] EASIS, "Deliverable D1.2-5 - Discussions and findings on Fault Tolerance", EASIS Partnership, 2006, www.easis.org
- [FTS06] EASIS, "Deliverable D1.2-3 - Fault Tolerance Services", EASIS Partnership, 2006, www.easis.org
- [FxR02] H. Heinecke, A. Schedl, B. Hedenetz, "FlexRay - ein Kommunikationssystem für das Automobil der Zukunft", Elektronik Automotive, pp. 36-45, Sep-2002
- [GAN03] J. Ganssle, "Watching the Watchdog," embedded world, 2003
- [GDi02] G. Leen, D. Heffernan: "Expanding Automotive Electronic Systems", IEEE Computer, pp. 5-6, January 2002
- [Glo03] AUTOSAR, "AUTOSAR Glossary", 2003, www.autosar.org
- [Glo07] EASIS, "EASIS Glossary", 2007 http://www.easis-online.org/wEnglish/download/deliverables_WP0.shtml?navid=15
- [GTD06] EASIS, "Deliverable D1.2-10 - Data exchange concepts for gateways", EASIS Partnership, 2006, www.easis.org
- [GTS06] EASIS, "Deliverable D1.2-12 - Security and firewall concepts for gateways", EASIS Partnership, 2006, www.easis.org
- [Har02] F. Hartwich, T. Führer, B. Müller and R. Hugel, Robert Bosch GmbH, "Integration of Time Triggered CAN (TTCAN_TC)," SAE, 2002

- [Hed01] Hedenetz, B. "Entwurf von verteilten fehlertoleranten Elektronikarchitekturen in Kraftfahrzeugen", Dissertation, Eberhard-Karls-Universität Tübingen, 2001
- [HIL06] M. Hiller, X. Chen, et. al., "Dependability Services in the EASIS Software Platform", presented at DSN 2006 Workshop on Architecting Dependable Systems, 2006, http://www.easis-online.org/wEnglish/img/pdf-files/wads_2006_easis.pdf
- [HOS07] Wikipedia, "History of Operating System", 2007, http://en.wikipedia.org/wiki/History_of_operating_systems
- [HRK00] B. Hedenetz, J. Ruh, M. Kühlerwein, Th. Ringler, et. al., "OSEKtime - the new fifth OSEK/VDX group: A Dependable Fault-Tolerant Real-Time Operating System and Communication Layer for By-Wire Applications", presented at VDI Baden-Baden, 2000, pp. 59
- [HWA06] EASIS, "Deliverable D2.2 - Conceptual Hardware Architecture Specification", EASIS Partnership, 2006, www.easis.org
- [HWAA06] EASIS, "Appendix to Deliverable D2.2 - Conceptual Hardware Architecture Specification", EASIS Partnership, 2006, www.easis.org
- [HWV06] EASIS, "Deliverable D2.4 An ECU prototype with fault tolerance characteristics", EASIS Partnership, 2006, www.easis.org
- [IEC01] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 1: General requirements", 1998
- [IEC02] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems", 1998
- [IEC03] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 3: Software requirements", 1998
- [IEC04] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 4: Definitions and abbreviations", 1998
- [IEC05] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 5: Examples of methods for the determination of safety integrity levels", 1998
- [IEC06] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3", 1998
- [IEC07] IEC61508, "Functional safety of electronical/electronic/programmable electronic safety-related systems - Part 7: Overview of techniques and measures", 1998
- [IECFAQ] IEC61508, "IEC 61508 FAQs", <http://www.iec.ch/zone/fsafety/questions.htm>
- [ISO11898] International Organization for Standardization "ISO 11898: Road vehicles — Controller area network (CAN)", Geneva, 2003
- [ISO15] CMMI/ISO 15504: Information technology - (Software) Process assessment - Parts 1 to 5

- [ISO21] ISO/WD 26262-1, TC 22/SC3 N 035, Road vehicles – functional safety – part 1: Glossary, 2005-12-22.
- [ISO22] ISO/WD 26262-2, TC 22/SC3 N 007, Road vehicles – functional safety – part 2: Management of functional safety, 2005-09-26
- [ISO23] ISO/WD 26262-3, TC 22/SC3 N 008, Road vehicles – functional safety – part 3: Concept phase, 2005-09-26
- [ISO24] ISO/WD 26262-4, TC 22/SC3 N 009, Road vehicles – functional safety – part-4: Product development system, 2005-09-25
- [ISO25] ISO/WD 26262-5, TC 22/SC3 N 010, Road vehicles – functional safety – part 5: Product development hardware, 2005-10-7
- [ISO26] ISO/WD 26262-6, TC 22/SC3 N 011, Road vehicles – functional safety – part 6: Product development software, 2005-09-19
- [ISO27] ISO/WD 26262-7, TC 22/SC3 N 012, Road vehicles – functional safety – part 7: Production and operation, 2005-09-16
- [ISO28] ISO/WD 26262-8, TC 22/SC3 N 013, Road vehicles – functional safety – part 8: Supporting processes, 2005-09-26
- [ISO7498-1] International Organization for Standardization „ISO 7498-1: Information Technology – Open System Interconnection – Basic Reference Model: The Basic Model“, Standard, Geneva, 1994
- [Kop02] P. Koopman, "Dependability for Embedded Systems", Lecture, Carnegie Mellon, 2002
- [Kop04] P. Koopman, T. Chakravarty, "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks", presented at The International Conference on Dependable Systems and Networks, DSN-2004, 2004
- [Kru98] M. Krug, "Concept and Implementation of a Dependable Automotive Operating System", Dissertation in Computer of Science: University of Tübingen, 1998, pp. 57, Chapter 5: Defining a Fault Hypothesis
- [Lam80] L. Lamport; R.Shostak; M.Pease, "Reaching Agreement in the Presence of Faults," Journal of the ACM, vol. 27, no. 2, pp. 228..234, Apr-1980.
- [Lam82] L. Lamport, M. Pease, R. Shostak, "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems, vol. 4, pp. 382-401, 1982
- [LAN97] D. Lantrip, "General Purpose Watchdog Timer Component for a Multitasking System": embedded world, 1997, <http://www.embedded.com/97/feat29704.htm>
- [Lap95] J.-C. Laprie, Dependability - its attributes, impairments and means, Predictably Dependable Computing Systems, pp. 3-24, Springer-Verlag, 1995
- [Lim05] M. Limam, "Conception and Implementation of an Agreement Protocol for Fault-Tolerant Automotive Embedded Systems", Diploma thesis in Institut für Automatisierungs- und Softwaretechnik: University of Stuttgart, 2005

- [LIN2.0] LIN Consortium „LIN Specification Package“, Revision 2.0, Motorola, Munich, 2003
- [LSP80] Lamport L.; Shostak R.; Pease M.: Reaching Agreement in the Presence of Faults. In: Journal of the ACM, vol. 27, no. 2, pp. 228..234, April 1980
- [LSP82] Lamport L.; Shostak R.; Pease M.: The Byzantine Generals Problem, In: ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 382..401, July 1982
- [McK05] McKinsey&Company, "Automotive Electronics", McKinsey Automotive Electronics Initiative, 2005
- [McK06] Hoch, et. al., "The Race to Master Automotive Embedded Systems Development", McKinsey&Company, 2006
- [Nah02] O. Nahmsuk, P. Shirvani, E. McCluskey, "Control-Flow Checking by Software Signatures," IEEE Transaction on Reliability, vol. 51, pp. 111-121, Mar-2002
- [OSK01] OSEK, "OSEK/VDX Time-Triggered Operating System Specification 1.0", 2001 www.osek-vdx.org
- [OSK05] OSEK, "OSEK/VDX Operating System Specification 2.2.3", ISO 17356-3:2005, Road vehicles -- Open interface for embedded automotive applications -- Part 3: OSEK/VDX Operating System (OS), 2005, www.osek-vdx.org
- [Rei05] W. Reinelt, W. Klier, G. Reimann., "Systemsicherheit des Active Front Steering", in Automatisierungstechnik, 53(1):36-43, 2005
- [Ruh04] J. Ruh, "Entwurf von fehlertoleranten Steuergeräteapplikationen in Kraftfahrzeugen unter Berücksichtigung moderner Entwicklungsmethodiken", in Fakultät für Informatik der Eberhard-Karls-Universität Tübingen, 2004
- [RWT02] RWTH Aachen, Lecture Notes “Fault Tolerant Computer Systems” Summer semester 02, Lectures 9: Byzantine agreement, <http://www-i4.informatik.rwth-aachen.de/lufg/lvs/teaching/ss02/ftcs/index.html>
- [SCK04] C. Shelton, P. Charles, P. Koopman, "Improving System Dependability with Functional Alternatives", presented at The International Conference on Dependable Systems and Networks, DSN, 2004
- [SIV04] H. Sivencrona, "On the Design and Validation of Fault Containment Regions in Distributed Communication Systems", Dissertation in Department of Computer Engineering. Göteborg, Sweden: Chalmers University of Technology, 2004, pp. 165-185
- [SMT06] M. Jörg, "Dynamic Software Module Tests - A Process Component for Better Software Quality", in ATZ Elektronik, 2006, pp. 36
- [Spi07] K. Spisic, "Concept and Implementation of a Fault Management Framework for a dependable in-vehicle E/E-architecture", Diploma thesis in Lehrstuhl Informatik XI Software für eingebettete Systeme: RWTH Aachen, 2007
- [SWA06] AUTOSAR, "AUTOSAR Layered Software Architecture, v2.0", 2006 http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf

- [SWI06] EASIS, "Deliverable D1.3 A prototypical implementation of the common software platform", EASIS Partnership, 2006 www.easis.org
- [SWP06] EASIS, "Deliverable D1.2-0 - Overall Description of the EASIS Software Platform", EASIS Partnership, 2006, www.easis.org
- [SWR07] EASIS, "Requirements Specification WP1 Software Architecture", EASIS Consortium, 2007, www.easis.org
- [SWS03] Bosch, "Steering Wheel Angle Sensor LWS3.6 Technical Customer Documentations", 2003
- [SWT06] EASIS, "Deliverable D1.2-1 - Software topology", EASIS Partnership, 2006, www.easis.org
- [SML06] M. Hause, A. Korff, "An Overview of SysML for Automotive Systems Engineers", ATZ-Elektronik, Mar-2007, pp. 22-29, 2007
- [Tin03] K. Tindell, F. Wolf, R. Ernst, "Safe Automotive Software Development, Chapter 2: The need for a protected OS in high integrity automotive systems", presented at Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03), IEEE Computer Society, Munich, Germany, Mar-2003
- [TMg02] T. Bertram, M. Torlo: Globale dynamische Fehlertoleranz – Systemverband als Basis für sichere X-by-Wire-Systeme, ATZ Automobiltechnische Zeitschrift 104, Nr. 11, 2002, 1008-1014
- [TRA06] V. Lauer, et. al., "EASIS – Electronic Architecture and System Engineering for Integrated Safety Systems", presented at Transport Research Arena Europe, 2006, http://www.easis-online.org/wEnglish/img/pdf-files/tra_2006_easis.pdf
- [TRV06] EASIS, "Deliverable D5.3 - Functional Test Report of EASIS Architecture Validator", EASIS Partnership, 2006, www.easis.org
- [μCI06] P. Sundaram, J. G. D'Ambrosio, "Controller Integrity in Automotive Failsafe System Architectures", presented at 2006 SAE World Congress, Detroit, Michigan, Apr-2006
- [VAL06] EASIS, "Specification of EASIS Validator with Telematics Gateway, WT5.1 Deliverable", EASIS Consortium, 2006, www.easis.org
- [Voa98] J. M. Voas, G. McGraw, Software Fault Injection: Inoculating Programs Against Errors: John Wiley & Sons, 1998

Appendix 6: Author's biography

Xi Chen was born in Wuhan, Hubei Province, China, on March 11, 1978. After two years of study in Applied Physics he came to Germany for his graduate study in 1998. He received his Diploma of Electrical and Computer Engineering and Master of Science degree in Information Technology at the [University of Stuttgart](#) in December 2003. In February 2004 he joined Daimler AG Group Research and Advanced Engineering as a research associate in the department of Electric/Electronic Architecture. In Spring 2004, he started pursuing his Ph.D. in the Faculty of Electrical and Computer Engineering in the [Institute für Technik der Informationsverarbeitung](#) at the [University of Karlsruhe \(TH\)](#). In February 2007 he joined ZF-Lenksysteme in Schwäbisch Gmünd as a Safety Manager in the Central Development of Functional Safety.

Appendix 7: Lebenslauf

Persönliche Daten

Familienname: Chen
 Vorname: Xi
 Geburtsdatum: 11. März 1978
 Geburtsort: Wuhan, China
 Familienstand: ledig
 E-Mail: xi.chen@gmx.de



Akademische Bildung

02/2004 – 02/2008 Promotion am Institut für Technik der Informationsverarbeitung der Universität Karlsruhe in der Zusammenarbeit mit der Zentralen Forschung der Daimler AG
 10/1999 – 12/2003 Doppelstudium in Elektrotechnik und Informationstechnologie an der Universität Stuttgart mit den Abschlüssen Dipl.-Ing. und Master of Science
 10/1998 – 10/1999 Intensiver Deutsch-Sprachkurs an der Universität Stuttgart
 10/1996 – 10/1998 Vier Semester Physikstudium an der Central China Normal University
 09/1984 – 07/1996 Schulbesuch in Wuhan, Erwerb der allgemeinen chinesischen Hochschulreife

Berufserfahrung

Seit 02/2007 Teilprojektleiter für die Entwicklung der Automobilsicherheit in den Kundenprojekten bei der [ZF-Lenkssysteme](#) GmbH in Schwäbisch Gmünd
 02/2004 – 01/2007 Projektarbeit während der Industriepromotion bei der Daimler AG

- Ingenieur in der Zentralen Forschung der Daimler AG - Fahrzeugkomfortelektronik
- Teilprojektleiter im EU-Projekt [EASIS](#) - Elektronikarchitektur and Entwicklungsprozess
- Mitarbeiter in der [AUTOSAR](#)-Konsortium - Arbeitsgruppe „Sicherheit“

 05/2003 – 12/2003 Industriediplomarbeit bei der [Daimler AG](#) in Esslingen zum Thema „Testautomatisierung der Steuergerätsoftware im Fahrzeug“
 11/2002 – 03/2003 Industriapraktikum und Werkstudententätigkeit bei [Sony International](#), Mobile Multimedia Lab
 03/1999 – 12/2003 Wissenschaftliche Hilfskraft beim

- Institut für elektrische und optische Nachrichtentechnik ([INT](#))
- Institut für Plasmaforschung ([IPF](#))
- Fraunhofer [Institut für Arbeitswirtschaft und Organisation](#) (IAO)

 08/1999 – 04/2000 Werkstudenten- und Ferientätigkeit bei der [Robert Bosch GmbH](#)

Soziales Engagement

12/2005 – 01/2007 Doktorandensprecher der Daimler AG
 Seit 09/2001 [VDE](#)-Mitglied der Hochschulgruppe und Young-Professional

Integrated Safety System, as one of the most promising automotive safety technologies, integrates safety functions across the domain boards and communication networks. In this dissertation, concepts of the electronic architecture are developed as a cooperative approach of engineering process, dependable hardware architecture and software platform.

The safety justified development process covers distributed rapid prototyping with virtual front-loading and correct-by-construction. Concepts of dependable hardware architecture are discussed with design guidelines of topology, ISS-application distribution and fault-tolerant ECU. The following dependability software services are designed: fault-tolerant communication with redundancy, end-to-end protection and Agreement Protocol, time partitioning and space partitioning for the application integration, fault management with dynamic reconfiguration, etc.