



## Assignment of master's thesis

<b>Title:</b>	A system for signals manipulation on the automotive ethernet
<b>Student:</b>	Bc. Oleksandr Korotetskyi
<b>Supervisor:</b>	Ing. Martin Štěpánek
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Software Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	until the end of summer semester 2023/2024

### Instructions

To test automotive control units, it is mandatory to simulate all the necessary values/ states of the input signals that are sent in Ethernet packets (frames). In some cases, it is easier to manipulate with data and simulate all the states directly in the packet than to use the simulation of other control units.

- 1) Perform research on signals and SAE levels in automotive ethernet
- 2) Perform research on possibilities of manipulation with data in ethernet packet and data security
- 3) Collect RQ for the test system
- 4) Design SW architecture
- 5) Design and implement SW for signal manipulation
- 6) Design a test strategy for the developed SW
- 7) Perform the test of implemented SW
- 8) Implementation should be done on Linux OS



Master's thesis

**A SYSTEM FOR SIGNAL  
MANIPULATION ON  
AUTOMOTIVE  
ETHERNET**

**Bc. Oleksandr Korotetskyi**

Faculty of Information Technology  
Department of Software Engineering  
Supervisor: Ing. Martin Štěpánek  
January 11, 2024

Czech Technical University in Prague  
Faculty of Information Technology

© 2023 Bc. Oleksandr Korotetskyi. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Korotetskyi Oleksandr. *A system for signal manipulation on Automotive Ethernet*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

# Contents

<b>Acknowledgments</b>	<b>ix</b>
<b>Declaration</b>	<b>x</b>
<b>Abstract</b>	<b>xi</b>
<b>Abbreviations</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Statement, Objectives & Methodology . . . . .	2
1.3 Thesis Outline . . . . .	2
<b>2 Preliminaries</b>	<b>5</b>
2.1 Highlights of Driving Automation . . . . .	6
2.1.1 Taxonomy of Driving Automation . . . . .	6
2.1.2 Functional Safety . . . . .	9
2.1.2.1 ASIL . . . . .	9
2.1.2.1.1 ASIL FTTL . . . . .	10
2.1.2.1.2 ASIL & Software Verification. . . . .	11
2.2 E/E Architecture . . . . .	12
2.2.1 E/E Domains . . . . .	13
2.2.2 Types of E/E Architecture . . . . .	14
2.3 Automotive Networking . . . . .	15
2.3.1 CAN . . . . .	16
2.3.1.1 High-speed CAN (CAN-C) . . . . .	17
2.3.1.2 Low-speed CAN (CAN-B) . . . . .	17
2.3.1.3 CAN-FD . . . . .	17
2.3.2 LIN . . . . .	18
2.3.3 FlexRay . . . . .	18
2.3.4 MOST . . . . .	18
2.3.5 Automotive Ethernet . . . . .	19
2.3.5.1 ISO/OSI Model . . . . .	19
2.4 AUTOSAR . . . . .	24
2.4.1 Details on AUTOSAR Communication . . . . .	26
2.4.1.1 Signal . . . . .	26
2.4.1.2 Signal Group . . . . .	27
2.4.1.3 Protocol Data Units . . . . .	28
2.4.1.4 I-PDU Group . . . . .	29
2.4.1.5 I-PDU Multiplexing . . . . .	29
2.4.1.6 I-PDU Transmission . . . . .	30
2.4.2 AUTOSAR XML . . . . .	31
2.5 Security of in-vehicle communication . . . . .	32
2.5.1 Security in context of ISO-OSI Model . . . . .	33
2.5.2 Security in context of AUTOSAR . . . . .	35
2.5.3 Security in context of Functional Safety . . . . .	37
2.6 Existing Solutions . . . . .	38
2.7 Conclusion . . . . .	39

<b>3</b>	<b>Requirements Synthesis</b>	<b>41</b>
3.1	Functionality Limitations . . . . .	41
3.2	User Requirements . . . . .	42
3.2.1	Use Cases . . . . .	43
3.3	Hardware Requirements . . . . .	44
3.4	Software Requirements . . . . .	44
3.4.1	Functional Requirements . . . . .	44
3.4.2	Non-functional Requirements . . . . .	45
3.5	Conclusion . . . . .	45
<b>4</b>	<b>Software Design &amp; Implementation</b>	<b>47</b>
4.1	Architectural Design . . . . .	47
4.2	Technologies & Libraries . . . . .	49
4.2.1	ObjectBox . . . . .	49
4.2.2	PF_RING . . . . .	50
4.2.3	PcapPlusPlus . . . . .	51
4.2.4	Libjson-rpc-cpp . . . . .	51
4.2.5	PyQt6 . . . . .	52
4.2.6	Lxml . . . . .	52
4.3	Detailed Design & Implementation . . . . .	53
4.3.1	Client-Server Interface . . . . .	53
4.3.2	Client . . . . .	57
4.3.2.1	Views . . . . .	59
4.3.2.1.1	GUI View. . . . .	59
4.3.2.1.2	Console View. . . . .	62
4.3.2.2	ARXML Parsing . . . . .	62
4.3.2.2.1	Schema. . . . .	63
4.3.2.2.1.1	Object Anchor. . . . .	63
4.3.2.2.1.2	Values. . . . .	63
4.3.2.2.1.3	Schema Template. . . . .	64
4.3.2.2.2	Parser. . . . .	65
4.3.2.2.2.1	Internal Data Representation. . . . .	65
4.3.2.2.2.2	ARXML Processing. . . . .	65
4.3.2.2.2.3	Path Handling. . . . .	66
4.3.2.2.2.4	Value Handling. . . . .	66
4.3.2.2.2.5	Object Handling. . . . .	66
4.3.2.2.2.6	Query Handling. . . . .	67
4.3.2.2.3	Schema Uploading. . . . .	68
4.3.2.2.4	Saving of Extracted Data. . . . .	69
4.3.2.3	Rule Creator . . . . .	70
4.3.2.4	Controller . . . . .	71
4.3.2.5	Model . . . . .	71
4.3.2.5.1	JSON-RPC Client. . . . .	71
4.3.2.5.2	Rule Representation. . . . .	71
4.3.2.5.3	Worker Object. . . . .	71
4.3.3	Server . . . . .	72
4.3.3.1	Common . . . . .	73
4.3.3.1.1	Logging. . . . .	74
4.3.3.1.2	Error handling. . . . .	74
4.3.3.1.3	Shared Resource. . . . .	75
4.3.3.1.4	Utility Functions. . . . .	75
4.3.3.2	I/O Interfaces . . . . .	75

4.3.3.3	System Logic . . . . .	76
4.3.3.3.1	Command-Line Arguments. . . . .	76
4.3.3.3.2	Initialization. . . . .	76
4.3.3.3.3	Execution. . . . .	77
4.3.3.3.4	Clean-up. . . . .	77
4.3.3.3.5	Auxiliary Functions. . . . .	77
4.3.3.4	Rule Representations . . . . .	77
4.3.3.5	Database . . . . .	78
4.3.3.5.1	Overview. . . . .	79
4.3.3.5.2	Concurrency Control. . . . .	79
4.3.3.5.3	Database Operations. . . . .	79
4.3.3.5.4	Exception Handling. . . . .	79
4.3.3.5.5	Initialization and Destruction. . . . .	79
4.3.3.5.6	Active Rule Management. . . . .	79
4.3.3.6	Packet Parser . . . . .	79
4.3.3.6.1	Overview. . . . .	79
4.3.3.6.2	PacketArrived Method. . . . .	80
4.3.3.7	JSON RPC Server . . . . .	81
4.3.3.8	Strategy Namespace . . . . .	82
4.3.3.8.1	Strategy. . . . .	82
4.3.3.8.2	Strategy Manager. . . . .	82
4.3.3.8.3	Modification Strategy. . . . .	84
4.3.3.8.4	Filtering Strategy. . . . .	86
4.3.3.8.5	Strategy Elements. . . . .	87
4.3.3.8.5.1	Auxiliary Functions. . . . .	87
4.3.3.8.5.2	PDU Examination. . . . .	87
4.3.3.8.5.3	CRC Calculation. . . . .	88
4.3.3.8.5.4	AES128-CMAC Computation. . . . .	89
4.4	Conclusion . . . . .	90
<b>5</b>	<b>Testing</b> . . . . .	<b>91</b>
5.1	Theory of Testing . . . . .	91
5.2	Formulation of Testing Strategy . . . . .	93
5.2.1	Client . . . . .	93
5.2.2	Server . . . . .	95
5.3	Functionality Verification & Validation . . . . .	96
5.3.1	Client . . . . .	96
5.3.1.1	Component & Structural Testing: ARXML Parser . . . . .	96
5.3.1.2	Non-Functional Testing: ARXML Parser . . . . .	97
5.3.1.3	GUI Testing: Heuristic Analysis . . . . .	101
5.3.1.4	GUI Testing: User-Involved Tests . . . . .	102
5.3.1.5	System Testing . . . . .	103
5.3.2	Server . . . . .	104
5.3.2.1	System Testing: Setup of Environment . . . . .	104
5.3.2.2	System Testing: Results . . . . .	105
5.4	Conclusion . . . . .	112
<b>6</b>	<b>Conclusion</b> . . . . .	<b>115</b>

## List of Figures

2.1	Schematic view of the driving task. . . . .	6
2.2	Fault reaction after fault detection. . . . .	10
2.3	Examples of typical ECU processing types per functional domains. . . . .	13
2.4	An illustration of the main types of E/E architectural philosophies and technologies used for interconnection of ECUs. . . . .	15
2.5	Example of distributed automotive network architecture. . . . .	17
2.6	ISO/OSI reference model and protocols used in context of Automotive Ethernet. . . . .	19
2.7	100BASE-T1 physical layers and its sublayers. . . . .	20
2.8	Ethernet frame structure at data link layer. . . . .	21
2.9	IPv6 packet structure at the network layer. . . . .	21
2.10	UDP packet structure. . . . .	22
2.11	TCP packet structure. . . . .	22
2.12	Example of TCP communication. . . . .	23
2.13	The IEEE 1722 packet format with example 1722 payloads. . . . .	24
2.14	AUTOSAR layered software architecture. . . . .	25
2.15	Overview of AUTOSAR software layers & appropriate modules in detail [45]. . . . .	26
2.16	Encapsulation of data (an SDU) by adding a header (the PCI) to form a Protocol Data Unit processed by a lower layer. . . . .	28
2.17	Internal structure of Protocol Data Unit within the packet payload. . . . .	28
2.18	Protocol Header Processing for Transmission (direction – down) & Reception (direction – up) by layer N. . . . .	29
2.19	Modules of possible AUTOSAR Ethernet stack . . . . .	30
2.20	The procedure of ARXML creation according to the AUTOSAR meta-model. . . . .	31
2.21	Protocol overview for Automotive Ethernet. . . . .	33
2.22	Layered automotive security approach and related mechanisms. . . . .	34
2.23	Integration of the SecOC BSW with CAN. . . . .	35
2.24	AUTOSAR SecOC message authentication and freshness value verification. . . . .	36
2.25	Freshness Value structure. . . . .	37
2.26	Secured I-PDU structure. . . . .	37
2.27	Security mechanisms applied during the signal transmission from receiver's (to the left) and sender's (to the right) perspective. . . . .	38
2.28	The multifunctional bus control unit FlexDevice-L. . . . .	39
4.1	Architecture of a system for signals manipulation on Automotive Ethernet. . . . .	48
4.2	ObjectBox CRUD Operations per second in comparison with SQLite. . . . .	50
4.3	PF_RING's architecture. . . . .	50
4.4	Generation of libjson-rpc-cpp stub classes and their usage. . . . .	52
4.5	Simplified UML sequence diagram, showcasing a successful internal process for ARXML parsing, acquiring of user data, rule creation and sending of UPSERT request to JSON-RPC server (GUI enabled). . . . .	58
4.6	Low-fidelity wireframe with the Signal Modification tab active. . . . .	59
4.7	Low-fidelity wireframe with the Traffic Filtering tab active. . . . .	60
4.8	Client GUI at the time of operation with Signal Modification tab active. . . . .	61



4.9	Client GUI at the time of operation with Traffic Filtering tab active. . . . .	62
4.10	Transformation of one old 'Frame' object into two new 'Frame' objects. . . . .	69
4.11	Internal principle of the system's server-side. . . . .	73
4.12	Specifications of 'common' classes and enumerations. . . . .	74
4.13	Specification of SystemLogic class. . . . .	76
4.14	Specification of Database class. . . . .	78
4.15	Application of strategies in multithread environment. . . . .	81
4.16	Featured compounds of stgy namespace. . . . .	83
4.17	Assumed L-PDU structure, the program is intended to work with. . . . .	85
5.1	Comparison of Waterfall (a) and V-Model (b) representations of a system's development lifecycle. . . . .	92
5.2	Dependency of ARXML processing time on the number of frames . . . . .	99
5.3	Dependency of ARXML processing time on the file size. . . . .	100
5.4	Dependency of ARXML processing time on the number of signals. . . . .	101
5.5	Testing setup. . . . .	104
5.6	Automotive Ethernet 1000BASE-T1 cable. . . . .	105
5.7	NETLion 1000 Media Converter. . . . .	105
5.8	Verification of successfully performed manipulation with three signals using CANoe. . . . .	106
5.9	Average Ethernet packet processing time on Computers A and B in Transparent Gateway mode, one capture thread utilized. . . . .	107
5.10	Average Ethernet packet processing time on Computers A and B in Traffic Filtering mode, one capture thread utilized. . . . .	109
5.11	Average Ethernet packet processing time on Computers A and B in Signal Modification mode (one signal modified), one capture thread utilized. . . . .	109
5.12	Average Ethernet packet processing time on Computers A and B in Signal Modification mode (one signal modified) with in-PDU CRC recalculated, one capture thread utilized. . . . .	110
5.13	Average Ethernet packet processing time on Computers A and B in Signal Modification mode with merely two in-PDU MACs recalculated, one capture thread utilized. . . . .	110
5.14	Average Ethernet packet processing time on Computers A and B in Signal Modification mode (ten signals modified within the same PDU) with ten in-PDU CRC recalculated, one capture thread utilized. . . . .	111
5.15	Average Ethernet packet processing time on Computers A and B in Signal Modification mode (ten signals modified within the same PDU), three capture thread utilized. . . . .	111
5.16	Average Ethernet packet processing time on Computers A and B in Signal Modification mode (ten signals modified within the same PDU) with ten in-PDU CRC recalculated, one capture thread utilized. . . . .	112

## List of Tables

2.1	Summary of levels of driving automation according to SAE J3016. . . . .	8
2.2	ASIL classification. . . . .	10
2.3	Methods for software unit verification. . . . .	11

2.4	Comparison of Automotive Protocols. . . . .	16
4.1	JSON RPC TEST method specification. . . . .	54
4.2	JSON RPC START method specification. . . . .	54
4.3	JSON RPC STOP method specification. . . . .	54
4.4	JSON RPC UPSERT method specification. . . . .	55
4.5	JSON RPC GET method specification. . . . .	56
4.6	JSON RPC SET method specification. . . . .	56
4.7	JSON RPC DELETE method specification. . . . .	57
4.8	Value locations. . . . .	63
4.9	Value formats. . . . .	64
5.1	Summary on component and structural testing of ARXML Parser abstract component. . . . .	97
5.2	Results of non-functional testing of ARXML Parser, sorted by processing time in ascending order. . . . .	98
5.3	User interface problems & defects identified by the heuristic analysis. . . . .	102
5.4	User interface problems & defects identified by user-involved testing. . . . .	103
5.5	Network interfaces throughput in the Transparent Gateway mode, one capture thread utilized. . . . .	106
5.6	Registered throughout of network interfaces depending on the number of capture threads, having 19 active rules for signal modification applied in the same packet. . . . .	108

## List of code listings

2.1	Fragment of plausible ARXML inner structure. . . . .	31
4.1	Schema in <code>.yaml</code> format used for ARXML parsing. . . . .	64
4.2	Generated frame specification in JSON format. . . . .	69
4.3	Schema in <code>.fbs</code> format used in the database representation of rules. . . . .	77

*First and foremost, I would like to express my gratitude to my academic advisor for his guidance throughout the writing of this thesis. Additionally, I wish to extend my sincere thanks to the Czech Technical University in Prague and Porsche Engineering for their support during some of the most challenging times of my life, and for imparting invaluable knowledge and experience. Last but not least, I would like to thank my family and Anna Radiuk for their unwavering support and for never leaving my side.*

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague, Czech Republic on January 11, 2024

.....

## Abstract

As the automotive industry undergoes a rapid transformation towards connected electric vehicles and autonomous driving technologies, the need for advanced and promising communication solutions like Automotive Ethernet becomes paramount. This study delves into the relationship between driving automation, vehicle electronic architecture, and automotive networking, emphasizing the significance of Automotive Ethernet. Significantly, it explores the feasibility of signal manipulation within the Automotive Ethernet network for potential facilitation of vehicle testing, addressing the challenges involved. Eventually, the research leads to the development and testing of a software system designed for subtle signal manipulation, equipped to bypass security mechanisms mandated by functional safety standards.

**Keywords** Automotive Ethernet, AUTOSAR, functional safety, E/E architecture, driving automation, electronic control unit, vehicle testing, software

## Abstrakt

Vzhledem k tomu, že automobilový průmysl prochází rychlou transformací směrem k propojeným elektrickým vozidlům a technologiím autonomního řízení, stává se nezbytnou potřebou pokročilá a slibná komunikační řešení jako je Automotive Ethernet. Tato studie se zabývá vztahem mezi automatizací řízení, elektronickou architekturou vozidel a automobilovým sítěním, s důrazem na význam Automotive Ethernetu. Podstatně zkoumá proveditelnost manipulace se signály v síti Automotive Ethernet pro potenciální usnadnění testování vozidel, přičemž řeší zahrnuté výzvy. Nakonec výzkum vede k vývoji a testování softwarového systému navrženého pro sofistikovanou manipulaci se signály, vybaveného schopností obejít bezpečnostní mechanismy, které jsou vyžadovány standardy funkční bezpečnosti.

**Klíčová slova** Automotive Ethernet, AUTOSAR, funkční bezpečnost, E/E architektura, automatizace řízení, elektronická řídicí jednotka, testování vozidel, software



## Abbreviations

<b>1000BASE-T</b>	1000 Mbps Baseband Ethernet over Twisted Pair
<b>100BASE-T</b>	100 Mbps Baseband Ethernet over Twisted Pair
<b>10BASE-T</b>	10 Mbps Baseband Ethernet over Twisted Pair
<b>AAF</b>	Audio Application Format
<b>ACC</b>	Adaptive Cruise Control
<b>ACID</b>	Atomicity, Consistency, Isolation, Durability
<b>ACK</b>	Acknowledgement
<b>ADAS</b>	Advanced Driving Assistance Systems
<b>ADC</b>	Analog to Digital Converter
<b>ADS</b>	Automated Driving System
<b>AEPRIL</b>	Automotive Ethernet Protocol Injection Logger
<b>AES</b>	Advanced Encryption Standard
<b>AES128</b>	Advanced Encryption Standard 128-bit
<b>AES128-CMAC</b>	Advanced Encryption Standard 128-bit Cipher-based Message Authentication Code
<b>AH</b>	Authentication Header
<b>API</b>	Application Programming Interface
<b>ARP</b>	Address Resolution Protocol
<b>ARXML</b>	AUTOSAR XML
<b>ASIC</b>	Application-Specific Integrated Circuit
<b>ASIL</b>	Automotive Safety Integrity Level
<b>AUTOSAR</b>	Automotive Open System Architecture
<b>AVB</b>	Audio Video Bridging
<b>BC</b>	Body & Comfort
<b>BSD</b>	Berkeley Software Distribution
<b>BSW</b>	Basic Software
<b>BZ</b>	Botschaftszähler (in-PDU Alive Counter)
<b>CAN</b>	Controller Area Network
<b>CAN-B</b>	Controller Area Network with Low-Speed Capability
<b>CAN-C</b>	Controller Area Network with High-Speed Capability
<b>CAN-FD</b>	Controller Area Network with Flexible Data
<b>CIA</b>	Confidentiality, Integrity, and Availability
<b>CMAC</b>	Cipher-based Message Authentication Code
<b>CMD</b>	Command
<b>CNT</b>	Counter
<b>COM</b>	Communication Module
<b>CPU</b>	Central Processing Unit
<b>CRC</b>	Cyclic Redundancy Check
<b>CRUD</b>	Create, Read, Update, Delete
<b>CS</b>	Checksum
<b>CSM</b>	Cryptographic Services Manager
<b>CSMA/CD</b>	Carrier Sense Multiple Access with Collision Detection
<b>CWR</b>	Congestion Windows Reduced
<b>DAS</b>	Driving Automation System
<b>DB</b>	Database
<b>DCM</b>	Diagnostic Communication Manager
<b>DDoS</b>	Distributed Denial of Service
<b>DDT</b>	Dynamic Driving Task
<b>DEI</b>	Drop Eligible Indicator
<b>DEM</b>	Diagnostic Event Manager
<b>DFS</b>	Data Frame Specifications
<b>DHCP</b>	Dynamic Host Configuration Protocol
<b>DIO</b>	Digital Input/Output
<b>DLC</b>	Data Length Code

<b>DLL</b>	Data Link Layer
<b>DoIP</b>	Diagnostics over Internet Protocol
<b>DoS</b>	Denial of Service
<b>DPDK</b>	Data Plane Development Kit
<b>DS</b>	Differentiated Services
<b>DTI</b>	Diagnostic Test Interval
<b>DTLS</b>	Datagram Transport Layer Security
<b>E/E</b>	Electrical & Electronic
<b>E2E</b>	End to End
<b>EA</b>	Enterprise Architect
<b>ECE</b>	Explicit Congestion Notification Echo
<b>ECN</b>	Explicit Congestion Notification
<b>ECU</b>	Electronic Control Unit
<b>EEPROM</b>	Electrically Erasable Programmable Read-Only Memory
<b>ETH</b>	Ethernet
<b>FCS</b>	Frame Check Sequence
<b>FD</b>	Flexible Data
<b>FIM</b>	Function Inhibition Manager
<b>FIN</b>	Finish
<b>FIPS</b>	Federal Information Processing Standards
<b>FR</b>	Functional Requirement
<b>FTDMA</b>	Flexible Time Division Multiple Access
<b>FTTI</b>	Fault Tolerance Time Interval
<b>FV</b>	Freshness Value
<b>GF</b>	Galois Field
<b>GNU</b>	GNU's Not Unix
<b>GPL</b>	General Public License
<b>GPS</b>	Global Positioning System
<b>GPT</b>	General Purpose Timer
<b>gPTP</b>	Generic Precision Time Protocol
<b>GPU</b>	Graphics Processing Unit
<b>GUI</b>	Graphical User Interface
<b>Hi-Fi</b>	High-fidelity
<b>HMI</b>	Human-Machine Interface
<b>HPCU</b>	High Performance Computing Unit
<b>HTTP</b>	Hypertext Transfer Protocol
<b>HW</b>	Hardware
<b>I-PDU</b>	Interaction Protocol Data Unit
<b>I/O</b>	Input/Output
<b>I2C</b>	Inter-Integrated Circuit
<b>ICMP</b>	Internet Control Message Protocol
<b>ICU</b>	Input Capture Unit
<b>ID</b>	Identifier
<b>IDE</b>	Integrated Development Environment
<b>IEC</b>	International Electrotechnical Commission
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>IMAP</b>	Internet Message Access Protocol
<b>IP</b>	Internet Protocol
<b>IPG</b>	Interpacket Gap
<b>IPsec</b>	Internet Protocol Security
<b>ISO</b>	International Organization for Standardization
<b>JASPAR</b>	Japan Automotive Software Platform and Architecture
<b>JSON</b>	JavaScript Object Notation
<b>JSON</b>	RPC JavaScript Object Notation Remote Procedure Call
<b>L-PDU</b>	Data Link Layer Protocol Data Unit
<b>LAN</b>	Local Area Network



<b>LDW</b>	Lane Departure Warning
<b>LGPL</b>	Lesser General Public License
<b>LIN</b>	Local Interconnect Network
<b>LKA</b>	Lane Keeping Assist
<b>Lo-Fi</b>	Low-fidelity
<b>LSB</b>	Least Significant Bit
<b>LTS</b>	Long-Term Support
<b>MAC</b>	Media Access Control
<b>MAC</b>	Message Authentication Code
<b>MACsec</b>	Media Access Control Security
<b>MCAL</b>	Microcontroller Abstraction Layer
<b>MCU</b>	Microcontroller Unit
<b>MDA</b>	Model-Driven Architecture
<b>MDI</b>	Medium-Dependent Interface
<b>MDIO</b>	Media Data Input/Output
<b>MDT</b>	Minimum Delay Time
<b>MII</b>	Media Independent Interface
<b>MIT</b>	Massachusetts Institute of Technology
<b>MOST</b>	Media Oriented Systems Transport
<b>MPE</b>	Manchester Phase Encoding
<b>MSB</b>	Most Significant Bit
<b>MTBF</b>	Mean Time Between Failures
<b>MTTF</b>	Mean Time To Failure
<b>MTTR</b>	Mean Time To Repair
<b>MVC</b>	Model-View-Controller
<b>N-PDU</b>	Network Layer Protocol Data Unit
<b>NDP</b>	Neighbor Discovery Protocol
<b>NFC</b>	Near-Field Communication
<b>NFR</b>	Non-functional Requirement
<b>NIC</b>	Network Interface Card
<b>NIST</b>	National Institute of Standards and Technology
<b>NoSQL</b>	Not Only Structured Query Language
<b>NP-hard</b>	Non-deterministic Polynomial-time hard
<b>NRZ</b>	Non-Return to Zero
<b>NRZ-5</b>	Non-Return to Zero, type 5
<b>NS</b>	Nonce Sum
<b>ODD</b>	Operational Design Domain
<b>OEDR</b>	Object and Event Detection and Response
<b>OSEK</b>	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
<b>OSI</b>	Open Systems Interconnection
<b>OTA</b>	Over The Air
<b>P2P</b>	Point-to-Point
<b>PAS</b>	Publicly Available Specification
<b>PC</b>	Personal Computer
<b>PCI</b>	Protocol Control Information
<b>PCP</b>	Priority Code Point
<b>PCS</b>	Physical Coding Sublayer
<b>PDU</b>	Protocol Data Unit
<b>PDU-R</b>	Protocol Data Unit Router
<b>PHY</b>	Physical Layer
<b>PMA</b>	Physical Medium Attachment
<b>PSH</b>	Push
<b>PSI5</b>	Peripheral Sensor Interface 5
<b>PTP</b>	Precision Time Protocol
<b>QCI</b>	Quality of Service Class Identifier

<b>QM</b>	Quality Management
<b>RAM</b>	Random Access Memory
<b>RBS</b>	Remaining Bus Simulation
<b>RFC</b>	Request for Comments
<b>RPC</b>	Remote Procedure Call
<b>RS-Box</b>	Round Substitution Box
<b>RST</b>	Reset
<b>RTE</b>	Run-Time Environment
<b>S-Box</b>	Substitution Box
<b>SAE</b>	Society of Automotive Engineers
<b>SD</b>	Service Discovery
<b>SDK</b>	Software Development Kit
<b>SDU</b>	Service Data Unit
<b>SecOC</b>	Secure Onboard Communication
<b>SFD</b>	Start Frame Delimiter
<b>SIL</b>	Safety Integrity Level
<b>SM</b>	State Manager
<b>SMTP</b>	Simple Mail Transfer Protocol
<b>SOME/IP</b>	Scalable service-Oriented Middleware over Internet Protocol
<b>SOTIF</b>	Safety Of The Intended Functionality
<b>SPI</b>	Serial Peripheral Interface
<b>SQL</b>	Structured Query Language
<b>SSL</b>	Secure Sockets Layer
<b>SW</b>	Software
<b>SWC</b>	Software Component
<b>SYN</b>	Synchronize
<b>TCP</b>	Transmission Control Protocol
<b>TDMA</b>	Time Division Multiple Access
<b>TLS</b>	Transport Layer Security
<b>TM</b>	Trademark
<b>TP</b>	Transport Protocol
<b>TPID</b>	Tag Protocol Identifier
<b>TSN</b>	Time-Sensitive Networking
<b>UC</b>	Use Case
<b>UDP</b>	User Datagram Protocol
<b>UML</b>	Unified Modeling Language
<b>URG</b>	Urgent
<b>USB</b>	Universal Serial Bus
<b>UTF-8</b>	Unicode Transformation Format - 8-bit
<b>VB</b>	Virtual Bus
<b>VFB</b>	Virtual Functional Bus
<b>VID</b>	VLAN Identifier
<b>ViWi</b>	Vehicle Wireless
<b>VLAN</b>	Virtual Local Area Network
<b>VPN</b>	Virtual Private Network
<b>VW</b>	Volkswagen
<b>WAN</b>	Wide Area Network
<b>XML</b>	eXtensible Markup Language
<b>XOR</b>	Exclusive OR
<b>YAML</b>	Yet Another Markup Language



facilitate the testing process of in-vehicle Automotive Ethernet networks, thereby potentially contributing to reliability and safety of modern vehicles.

## 1.2 Problem Statement, Objectives & Methodology

This work addresses a broad spectrum of topics within the domain of in-vehicle networking and driving automation.

The overarching problem statement involves understanding and analysis of the relation between advanced automotive networking technologies (Automotive Ethernet), vehicular electronic & electrical architectures and the driving automation. Additionally, to assess the feasibility of executing accurate and real-time signal manipulation on an Automotive Ethernet network, this study incorporates the development of software designed for this specific capability.

The very objectives of this research are:

1. To provide a comprehensive overview of the classification of driving automation levels and their relation to vehicular functional safety.
2. To evaluate the role of vehicular electronic and electrical architectures in driving automation, dissecting the internal organization of electronic control units.
3. To generally examine the in-vehicle networking, with a focus on Automotive Ethernet, its role in driving automation, its operational principles and security aspects.
4. To explore the potential of signal manipulation within Automotive Ethernet in enhancing the vehicle testing process, including a review of existing tools for traffic manipulation in automotive networks.
5. To investigate AUTOSAR's contribution to in-vehicle communication and its integration with Automotive Ethernet.
6. To design the software capable of modifying signals within the Automotive Ethernet network while maintaining transparency to connected parties, ensuring the circumvention of security mechanisms primarily required by functional safety.
7. To establish a set of functional and non-functional requirements for the software, followed by its implementation and testing in a real laboratory environment to evaluate its effectiveness & applicability in various use cases, generally assessing the feasibility of manipulating signals on Automotive Ethernet.

Upon accomplishing the outlined objectives, the theoretical component of this work is intended to serve, to some extent, as an introduction to the world of automotive technologies, while the practical one could be considered as a technical monograph detailing the development of software and its subsequent testing.

The methodology for this thesis includes a combination of theoretical research, practical experimentation. This involves a detailed review of current standards and technologies, development of a software system for signal manipulation, empirical simulation-based testing in controlled environments and, to a certain degree, a statistical interpretation of achieved results.

## 1.3 Thesis Outline

This thesis is organized into six chapters, each showing the progression of the research and building upon the knowledge and findings of the previous ones:

**Chapter 1: Introduction** — This opening chapter sets the context for the research. It discusses the motivation behind the study, outlines the primary challenges and goals, and explains the methodology used. The chapter concludes by providing an overview of the thesis structure.

**Chapter 2: Preliminaries** — Here, foundational concepts related to driving automation are discussed, namely the taxonomy of driving automation. Next, the thesis introduces selected aspects of functional safety, outlining the potential utilization of the developed software within the vehicle testing process. Additionally, various aspects of vehicular electronic and electrical architecture are analyzed with focus on the inner organization of electronic control units. Later, the chapter explores different automotive networking technologies, concentrating on Automotive Ethernet, and proceeds with revealing the role of AUTOSAR in the in-vehicle communication process. Lastly, the security mechanisms adopted in automotive networking are reviewed and the conclusions are made.

**Chapter 3: Requirements Synthesis** — In this chapter, the focus shifts to identifying and compiling the requirements for the resulting software. It covers the functional limitations of the system to develop, shaping the achievable functionality according to the theoretical provisions acquired in Chapter 2. Consequently, user expectations are described and both hardware and software requirements needed for the proposed system are defined in collaboration with one of the leading automotive companies. The chapter concludes by summarizing the work done.

**Chapter 4: Software Design & Implementation** — This technical chapter delves into the design and development of the proposed system according to the requirements defined in Chapter 3. It describes the suggested architectural design, justifies the utilized technologies & libraries, and provides detailed insights into the system's implementation. The chapter concludes with an evaluation of how the design and implementation meet the set requirements, additionally summarizing the benefits and drawbacks of the resulting implementation.

**Chapter 5: Testing** — The testing phase of the previously developed system is covered in this chapter. It begins with a discussion on the theoretical aspects of software testing, followed by the formulation of a testing strategy. The chapter generally describes the testing setup and presents the verification & validation of the *featured* system functionality through various tests. It concludes with a summary of the testing outcomes.

**Chapter 6: Conclusion** — The final chapter concludes the thesis. It summarizes the performed activities and key findings, discusses the implications of the work done, and suggests potential areas for future research.



# Preliminaries

This chapter progressively unravels the multifaceted aspects of driving automation, laying a comprehensive foundation for understanding the role of Automotive Ethernet and inner principles of its organization, with focus on possibilities of data manipulation and circumvention of existing security mechanisms.

First, a detailed classification of driving automation is presented (see section 2.1.1, differentiating levels based on the interaction between humans and machines. Then, the concept of functional safety is introduced in section 2.1.2, focusing on automotive security integrity levels and the inherent fault tolerance time interval, crucial for safety standards in automotive software development & testing. Moreover, the general applicability of real-time interference in traffic is outlined in this context (refer to paragraph 2.1.2.1.2).

Next, the nuances of electrical/electronic architecture of vehicles are examined in section 2.2, highlighting the organization of electronic control units into domain-specific functionalities and analyzing architectural models essential for higher automation levels.

Consequently, section 2.3 generally covers a topic of automotive networking, comprising a review of networking technologies and their qualitative characteristics; the role of Automotive Ethernet in highly automated vehicles is emphasized. The focus then narrows to its structure and the role of the ISO/OSI model in communication, providing insights into the each OSI model layer and featured protocols (refer to section 2.3.5.1).

AUTOSAR framework analysis follows in section 2.4, exploring the organization of software units within a car, principles of their connectivity and possibilities of real-time data manipulation in Automotive Ethernet. It includes a detailed look at the communication of software components and abstract data units such as signals and protocol data units, emphasizing their properties, structure and selected individual stages of their transmission.

Later, the security aspects of in-vehicle networking are examined in section 2.5 from the perspectives of the ISO/OSI model, AUTOSAR, and functional safety. Various security measures are analyzed, outlining the theoretical foundation for their circumvention.

Additionally, a brief review of existing solutions for real-time data manipulation in in-vehicle networking is provided in section 2.6, summarizing current advancements and implementations in the field.

The chapter concludes by reaffirming the feasibility of manipulation with signals in the Automotive Ethernet network, underscoring the critical requirements for successful execution and highlighting potential bottlenecks.

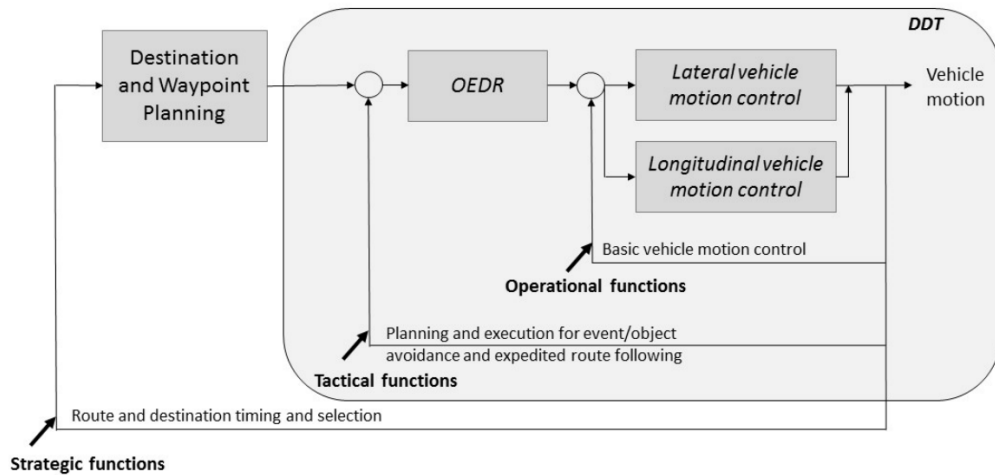
## 2.1 Highlights of Driving Automation

Driving automation is a cutting-edge area of development for the world's leading car manufacturers. To understand the role of the Automotive Ethernet in the development of partially and/or fully autonomous cars, it is necessary to consider the very concept of car autonomy and the resulting functional safety issues.

### 2.1.1 Taxonomy of Driving Automation

The topic of driving automation is closely related to the process of *driving* itself. Driving entails a variety of decisions and actions, which may or may not involve a vehicle being in motion. The overall act of driving task is schematically depicted in Figure 2.1 and can be divided into three types of driver effort:

- *Strategic* effort involves trip planning, such as deciding whether, when and where to go, how to travel, best routes to take, etc.
- *Tactical* effort involves maneuvering the vehicle in traffic during a trip, including deciding whether and when to overtake another vehicle or change lanes, selecting an appropriate speed, checking mirrors, etc.
- *Operational* effort involves split-second reactions that can be considered pre-cognitive or innate, such as making-micro-corrections to steering, braking and accelerating to maintain lane position in traffic or to avoid a sudden obstacle or hazardous event in the vehicle's pathway.



■ **Figure 2.1** Schematic view of the driving task [3].

A self-driving car, also known as an *autonomous car*, is a car that is capable of traveling without human input [4]. Self-driving cars use sensors to perceive their surroundings, such as optical and thermographic cameras, radar, lidar, ultrasound/sonar, GPS, odometry and inertial measurement units [5]. Also, further technologies used to achieve autonomous driving might include several forms of artificial intelligence [6].

SAE J3016 [3] defines 6 levels of automation, sketching an incremental evolution from no automation to fully autonomous vehicles [7]. Central to this taxonomy are the respective roles of the (human) *user* and the *driving automation system* (DAS) in relation to each other. Since changes in the functionality of a *driving automation system* change the role of the (human) user, they provide a basis for categorizing such system features:



- If the driving automation system performs the sustained longitudinal and/or lateral vehicle motion control subtasks of the DDT <sup>1</sup>, the driver does not do so, although s/he is expected to complete the DDT. This division of roles corresponds to Levels 1 and 2.
- If the driving automation system performs the entire DDT, the user does not do so. However, if a DDT fallback-ready user is expected to take over the DDT when a DDT performance-relevant system failure occurs or when the driving automation system is about to leave its *operational design domain* (ODD), which is specified by manufacturer. Then the user is expected to be receptive and able to resume DDT performance when alerted to the need to do so. This division of roles corresponds to Level 3.
- Lastly, if a driving automation system can perform the entire DDT and DDT fallback either within a prescribed ODD (Level 4) or in all driver-manageable on-road operating situations (Level 5) then any users present in the vehicle while the ADS is engaged are passengers.

Additionally, although the vehicle fulfills a role in this driving automation taxonomy, it does not change the role of the user in performing the DDT. By contrast the role played by the driving automation system complements the role of the user in performing the DDT, and in that sense changes it. In this way, according to [3], driving automation systems are categorized into levels based on:

1. Whether the driving automation system performs either the longitudinal or the lateral vehicle motion control subtask of the DDT.
2. Whether the driving automation system performs both the longitudinal and the lateral vehicle motion control subtasks of the DDT simultaneously.
3. Whether the driving automation system also performs the OEDR subtask of the DDT.
4. Whether the driving automation system also performs DDT fallback.
5. Whether the driving automation system is limited by an ODD.

Table 2.1 summarizes the six levels of driving automation in terms of these five elements. It is worth mentioning, that SAE's levels of driving automation are descriptive and informative, rather than normative, and technical rather than legal. Elements indicate minimum rather than maximum capabilities for each level [3].

In this table, 'system' refers to the driving automation system or ADS, as appropriate; definitions of some terms that seem to be obvious are omitted for the sake of brevity. In addition, as it was implicitly mentioned earlier, the DDT does not include strategic aspects of the driving task, such as determining destination(s) and deciding when to travel.

Apart from driving automation systems, driver assistance systems (DAS) exist and they support the driver in their primary driving task [8]. They inform and warn the driver, provide feedback on driver actions, increase comfort and reduce the workload by actively stabilising

---

<sup>1</sup>All of the real-time operational and tactical functions required to operate a vehicle in on-road traffic, excluding the strategic functions such as trip scheduling and selection of destinations and waypoints, and including, without limitation, the following subtasks:

1. Lateral vehicle motion control via steering (operational).
2. Longitudinal vehicle motion control via acceleration and deceleration (operational).
3. Monitoring the driving environment via object and event detection, recognition, classification, and response preparation (operational and tactical).
4. Object and event response execution (operational and tactical).
5. Maneuver planning (tactical).
6. Enhancing conspicuity via lighting, sounding the horn, signaling, gesturing, etc. (tactical)

or manoeuvring the car. Therefore, their responsibilities mostly overlap with typical driving automation systems (in some sense, these are synonyms); they assist the driver and do not take over the driving task completely, thus the responsibility always remains with the driver.

	Level	Name	Narrative Definition	DDT		DDT Fallback	ODD
				Sustained Lateral and Longitudinal Vehicle Motion Control	OEDR		
<b>Driver Performs Part or All of the DDT</b>							
	0	<i>No Driving Automation</i>	The performance by the driver of the entire DDT, even when enhanced by ASSs.	Driver	Driver	Driver	n/a
Driver Support	1	<i>Driver Assistance</i>	The sustained and ODD-specific execution by a DAS of either the lateral or longitudinal vehicle motion control subtask of the DDT (but not both simultaneously) with the expectation that the driver performs the remainder of the DDT.	Driver and System	Driver	Driver	Limited
	2	<i>Partial Driving Automation</i>	The sustained and ODD-specific execution by a DAS of both the lateral and longitudinal vehicle motion control subtasks of the DDT with the expectation that the driver completes the OEDR subtask and supervises the DAS.	System	Driver	Driver	Limited
<b>ADS Performs the Entire DDT (While Enabled)</b>							
Automated Driving	3	<i>Conditional Driving Automation</i>	The sustained and ODD-specific performance by an ADS of the entire DDT with the expectation that the DDT fallback-ready user is receptive to ADS-issued requests to intervene, as well as to DDT performance-relevant system failures in other vehicle systems, and will respond appropriately.	System	System	Fallback-Ready User (becomes the driver during the fallback)	Limited
	4	<i>High Driving Automation</i>	The sustained and ODD-specific performance by an ADS of the entire DDT and DDT fallback without any expectation that a user will need to intervene.	System	System	System	Limited
	5	<i>Full Driving Automation</i>	The sustained and unconditional (i.e., not ODD-specific) performance by an ADS of the entire DDT and DDT fallback without any expectation that a user will need to intervene.	System	System	System	Unlimited

■ **Table 2.1** Summary of levels of driving automation according to SAE J3016.

Advanced driving assistance systems (ADAS) are a subset of the driver assistance systems. ADAS are characterized by *all* of the following properties [9]:

- detect and evaluate the vehicle environment
- direct interaction between the driver and the system
- support the driver in the primary driving task

- provide active support for lateral and/or longitudinal control with or without warnings
- use complex signal processing

With respect to the aforementioned categories of driving tasks, nowadays ADAS are mainly focussing on the manoeuvring level. While ADAS are not autonomous driving systems, they play an important role in preparing cars for full autonomy [8]. Many of the features that make up a SAE Level I or Level II systems are made possible by ADAS.

For example, adaptive cruise control (ACC) and lane keeping assist (LKA) would not be possible without sensors to detect objects around the car. Similarly, the ADAS feature lane departure warning (LDW) system would not be effective without cameras or other sensors that can track the car's position on the road.

Consequently, the flawless operation of ADAS/DAS components inside a car becomes an unconditional requirement for ensuring the safety of a driver.

## 2.1.2 Functional Safety

In context of driving automation, ensuring the safety and reliability of software-driven systems within vehicles stands as a paramount concern. While, for instance, a system reliability can be evaluated using the mean time to failure (MTTF), mean time between failures (MTBF), and mean time to repair (MTTR) as failure metrics [10], the system safety is a more complex concept that encompasses a variety of factors.

The pursuit of automotive safety has led to the development and adoption of rigorous standards and methodologies, one of which is the ISO/PAS 21448:2019 standard. It addresses the safety of the intended functionality (SOTIF) of a system, by dealing with safety issues that arise because of functional insufficiencies, performance limitations and foreseeable misuses. Another important international standard in this context is ISO 26262, which addresses functional safety in road vehicles.

The objective of functional safety is 'freedom from unacceptable risk of physical injury or of damage to the health of people either directly or indirectly (through damage to property or to the environment) by the proper implementation of one or more automatic protection functions (often called safety functions)' [11]. The goal of the respective ISO 26262 specifications is to reduce safety risks in cars to a minimum. ISO 26262 specifically deals with the functional safety of electrical and electronic systems within a vehicle. In terms of software development & testing of the software-driven automotive systems, ISO 26262 compliance reduces the likelihood of accidents caused by *software-related* faults.

### 2.1.2.1 ASIL

Automotive Safety Integrity Level (ASIL) integral to the ISO 26262, plays a pivotal role in evaluating and mitigating risks associated with software components in vehicles. It is an adaptation of the Safety Integrity Level (SIL) used in IEC 61508 [12] for the automotive industry.

The primary purpose of ASIL is to categorize and assess safety requirements for software and electronic systems utilized in automobiles [11]. It offers a systematic framework for evaluating potential risks associated with these systems, ensuring that appropriate safety measures are meticulously integrated.

The standard delineates four ASILs: ASIL A, ASIL B, ASIL C, and ASIL D. Of these, ASIL D imposes the most stringent integrity & safety requirements on the product, while ASIL A represents the lowest threshold of safety criticality. Additionally, hazards categorized under Quality Management (QM) are exempt from the imposition of safety requirements [13].

The very classification is determined based on an extensive risk analysis that generally considers factors such as the system's function, potential impact on safety, probability of failure, and

the ability to detect and mitigate failures. To be precise, Table 2.2 summarizes the automotive safety integrity levels based on:

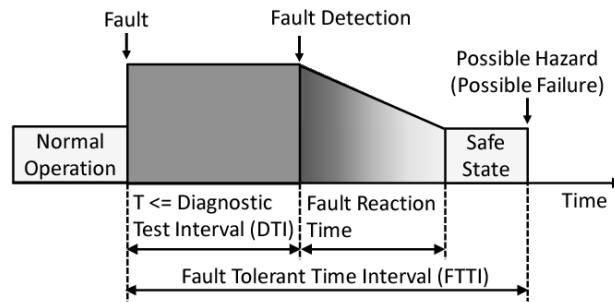
- **Severity** - potential severity of injuries caused by a hazardous event.
- **Exposure** - frequency of conditions that would potentially cause injury.
- **Controllability** - likelihood that the driver could act to prevent injury.

Severity		Exposure	Controllability		
			C1 (Simple)	C2 (Normal)	C3 (Difficult / Uncontrollable)
S1	<i>Light and moderate injuries</i>	E1 (Very low)	QM	QM	QM
		E2 (Low)	QM	QM	QM
		E3 (Medium)	QM	QM	A
		E4 (High)	QM	A	B
S2	<i>Severe and life threatening injuries</i>	E1 (Very low)	QM	QM	QM
		E2 (Low)	QM	QM	A
		E3 (Medium)	QM	A	B
		E4 (High)	A	B	C
S3	<i>Life threatening injuries, fatal injuries</i>	E1 (Very low)	QM	QM	A
		E2 (Low)	QM	A	B
		E3 (Medium)	A	B	C
		E4 (High)	B	C	D

■ **Table 2.2** ASIL classification [11].

Despite the assignment of QM and other ASILs is not directly related to SAE *levels* of driving automation, it is coherent with the very DAS/ADAS *functionality* indirectly: the higher responsibility is undertaken by a DAS/ADAS system at the time of driving, the higher potential risks shall be considered in case of its failure, implying the assignment of higher ASILs to electronic control units comprising it.

**2.1.2.1.1 ASIL FTTL.** Each ASIL carries specific safety requirements and measures (e.g. redundancy), impacting the development, testing and utilization processes. Higher ASILs mandate more stringent safety measures on fault tolerance, decreasing the overall acceptable *diagnostic test interval* (DTI) and *fault reaction time* to a few milliseconds (5-100 ms for higher ASILs) depending on the component requirements [14].



■ **Figure 2.2** Fault reaction after fault detection [15].

In the context of this thesis, it practically implies that any intervention in in-vehicle communication must occur more swiftly than the specified DTI (FTTI respectively<sup>2</sup>) to maintain transparency, by prevention of significant communication delays and the consequent misidentification of transmitted messages as erroneous or obsolete.

**2.1.2.1.2 ASIL & Software Verification.** As the complexity of software increases, so does the difficulty in testing it, especially when approaching real operating conditions. Table 2.3 specifies ASIL recommendations on usage of methods utilized for software unit verification.

Methods		ASIL			
		A	B	C	D
1	Walk-through	++	+	o	o
2	Pair-programming	+	+	+	+
3	Inspection	+	++	++	++
4	Semi-formal verification	+	+	++	++
5	Formal verification	o	o	+	+
6	Control flow analysis	+	+	++	++
7	Data flow analysis	+	+	++	++
8	Static code analysis	++	++	++	++
9	Static analyses based on abstract interpretation	+	+	+	+
10	Requirements-based test	++	++	++	++
11	Interface test	++	++	++	++
12	Fault injection test	+	+	+	++
13	Resource usage evaluation	+	+	+	++
14	Back-to-back comparison test between model and code, if applicable	+	+	++	++

■ **Table 2.3** Methods for software unit verification [11].

► **Note 2.1.** In Table 2.3, methods marked with '++' are highly recommended, marked with '+' are recommended and with 'o' have no recommendation for or against its usage for the identified ASIL.

In practice, the conduction of testing for in-vehicle modules, particularly for those responsible for DAS/ADAS functionalities, is a costly and complicated endeavor. This is especially evident when it comes to verifying the anticipated behavior of specific components in the course of system-level testing<sup>3</sup>. Therefore, sometimes it is cheaper and faster to *reproduce* inter-communication of inner software components, than to completely simulate all aspects of their behaviour in the real-time *XIL*<sup>4</sup> testing process. It becomes extremely relevant when applying ASIL recommended testing methods, during the system testing, listed below:

- **Interface test:** Interface testing, in the context of ISO 26262, focuses on verifying that the interfaces between various hardware and software components, such as ECUs, sensors, actuators, and communication networks, operate as expected to ensure functional safety. ISO 26262 emphasizes the importance of creating clear and detailed interface specifications [11]. This documentation should define the expected behavior of each interface, including the data format, data rates, timing constraints, error handling mechanisms, and fault tolerance requirements. This is because, every software unit may have been tested individually but their communication with other units is also very critical.

Interface testing ensures that all connected components are compatible with each other. This involves verifying that the signals transmitted and received by different ECUs match the

<sup>2</sup>Henceforth, for the sake of brevity, the use of these terms will be regarded as *equivalent* and *interchangeable*.

<sup>3</sup>Refer to section 5.1 for definition.

<sup>4</sup>XIL refers to a set of automotive testing methodologies including SIL (*software-in-the-loop*), HIL (*hardware-in-the-loop*) and MIL (*model-in-the-loop*), used for integrated testing of vehicle components.

defined interface specifications. Interface testing is performed to detect early failures that could go unnoticed till the integration testing process because at that stage, it would be even difficult to localize the issue.

Interface testing goes beyond functional correctness; it also assesses the system's behavior under various safety-critical conditions, such as fault scenarios. This includes testing the system's response to faults, such as signal corruption, ECU failures, or network interruptions.

- **Requirements based test:** Generally, this method is used to verify if the developed software matches the initial requirements [11]. In this method, the input values are derived from the behavioral requirements which are usually developed previously. If there are no documented requirements, they need to be deduced from the functional model as it has the executable specifications.

This method helps in systematic identification of implementation failures [11]. The failures occurring due to incomplete and inconsistent requirement are also identified.

- **Fault injection test:** According to ISO 26262 [11], a fault injection test uses special means to introduce faults into the test object during runtime. Despite the general applicability of this method in different contexts, usually it can be done within the software via a special test interface or specially prepared hardware.

In the contexts of software testing, software unit verification and software integration testing fault injection test means to modify the tested software unit (e.g. introduce faults into the software). Such modifications include injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, by corrupting software interfaces and by corrupting values of CPU registers or calibration parameters). The method is often used to improve the test coverage of the safety requirements (because during normal operation safety mechanisms are not invoked) and, in particular, to test the correctness of hardware-software interface related to safety mechanisms. Fault injection can also be used to verify freedom from interference [16].

Moreover, similar approach is described in ISO/PAS 21448:2019 [17], being mentioned as '*Injection of system inputs that trigger the potentially hazardous behaviour*'. According to the standard, in some cases, it is possible to emulate a potentially hazardous behaviour of the sensor by means of error (noise) injection at the simulation level.

Summarizing, the application of aforementioned tests commonly implies the assertion of behavior of particular in-vehicle module depending on the received input values. In context of this thesis, the input value itself is identified as an incoming signal transmitted over the Automotive Ethernet<sup>5</sup>. Proper '*on-the-way*' manipulation with incoming signals represents a uniform solution for simulation of different kinds of input values.

## 2.2 E/E Architecture

Almost all aspects of *electrical & electronic* (E/E) system development of a vehicle, including technical approaches, specified requirements, decision making about design structure, developments methods and potentially supported levels of driving automation, are affected by its E/E architecture. Vehicle architecture can be seen from various perspectives.

The *physical* vision illustrates the positioning and the connection of the used elements in the car such as ECUs, sensors, actuators, gateways, power supply, and switches. Moreover, it includes communication networks, wiring harness placement, and power distribution setup.

Another perspective of E/E architecture is *logical* vision; it focuses on the interconnection an interaction among various components and elements integrated into a car. Based on this point of

<sup>5</sup>Refer to sections 2.4.1.1 and 2.3.5 for further information about signals or Automotive Ethernet respectively.

view, the E/E architecture is responsible for proper data exchange, signal flows, communication and interface protocols.

Achievement of higher levels of driving automation necessitates the establishment of advanced and effective vehicle E/E architecture. This implies supplementing of E/E architectures with means to provide a driver not only with non-critical functionalities and features, but also with ADAS functionalities to have safe and more comfortable driving experiences. Accordingly, the automotive E/E architecture has advanced significantly, from various sensors and actuators to more powerful computing units to process a huge amount of data coming from the sensors for critical and non-critical functionalities [18].

### 2.2.1 E/E Domains

Carmakers distinguish several domains for embedded electronics in a car, even though sometimes the membership of only one domain for a given compartment is not easily justifiable. A domain is defined as 'a sphere of knowledge, influence, and activity in which one or more systems are to be dealt with (e.g., are to be built)' [19].

Historically, five domains were identified: *Powertrain*, *Chassis*, *Body & Comfort*, *HMI (Infotainment)*, and *Telematics*. However, currently *ADAS* and/or *ADAS sensors* domain(s) is being additionally distinguished [20] due to its general impact on the automotive industry & driver experience.

The Powertrain Domain encompasses systems integral to the vehicle's longitudinal movement, such as the engine, transmission, and their associated components. In the Chassis Domain, the focus is on the vehicle's wheel system, primarily addressing steering and braking mechanisms. The Body Domain covers components external to vehicle dynamics, which facilitate user support features, including but not limited to airbags, windshield wipers, internal lighting, window mechanisms, air conditioning, and seating arrangements. The Infotainment entails devices facilitating communication between the vehicle's electronic systems and the driver, including display panels and control switches. The Telematics Domain is concerned with systems enabling the vehicle's communication with external networks, covering functionalities like radio connectivity, navigation, internet access, and electronic payments. Lastly, the ADAS Domain aims to augment vehicular safety by assisting the driver and potentially automating certain driving operations. This is achieved through continuous monitoring of the vehicle's external environment and the driver's behavior.

Domain	Control Loop Time	Realtime	ASIL	Processing Type	Software type	Examples
Infotainment	ms	AVB, soft real time	Mostly QM, Up to B	$\mu$ C with GPU	Linux/ Android/ RTOS (AUTOSAR Com)	Touch Screen, Media, Cluster
Body & Comfort	ms	Soft real time	Mostly QM, Up to B	$\mu$ P	AUTOSAR	Doors, Seats, Locks, HVAC
Powertrain	$\mu$ s	Hard real time	Up to D	$\mu$ P Multi-core	AUTOSAR	Engine, Transmission, Motor, Inverter, DCDC
Chassis	ms/ $\mu$ s	Hard real time	Up to D	$\mu$ P Multi-core	AUTOSAR	ABS/ESC, Suspension
ADAS Domain	ms	Hard real time	Up to D	$\mu$ C with GPU	AUTOSAR + Linux/ RTOS	ADAS Domain (Primary & Secondary)
ADAS Sensors	ms	Hard/Soft Real time	Up to D (B & C common)	$\mu$ C with GPU	AUTOSAR + Linux/ RTOS/ (FPGA)	Camera, Lidar, Radar

■ **Figure 2.3** Examples of typical ECU processing types per functional domains [20].



Electronic systems across different vehicle domains exhibit distinct characteristics. The powertrain and chassis domains require significant computational power and adhere to strict real-time constraints, with the chassis domain featuring a more distributed hardware architecture. The telematics domain demands high data throughput. Increasing automation levels, particularly in the ADAS domain, escalate the computational needs of vehicle ECUs.

### 2.2.2 Types of E/E Architecture

In consequence to the domain diversity, the organization, connectivity and primary functions of car ECUs differ. This diversity extends to varied technological solutions, encompassing communication networks, embedded software design techniques, and verification approaches [20].

It is common in today's vehicles that the electronic architecture includes different types of networks interconnected by gateways and signals are exchanged by 70 to 150 ECUs in order to manage the car, including its software system. Depending on the architecture, the number of ECUs could be reduced considerably by merging various mixed-critical applications into one multi-core ECU [21].

The organization of ECUs in vehicles typically begins with *function-specific* ECUs, where each ECU is dedicated to a distinct function [20]. The next level is *domain-specific* ECUs, which manage a group of functions within a particular domain, often involving high computation and numerous input/output (I/O) devices. These ECUs interconnect with multiple function-specific units. In contrast, *zonal* ECUs distribute data and power throughout the vehicle, supporting features in their specific zones [22]. They act as gateways, switches, and smart junction boxes, interfacing with various sensors, actuators, and displays.

The overall design of E/E architectures can be split in several several approaches denoted at Figure 2.4, differing in cost, complexity and effectiveness:

- **Distributed E/E Architecture**, or **Multi-Bus Gateway Architecture**, integrates the function-specific ECUs with a central gateway connected via a CAN bus<sup>6</sup> [23]. Utilization of the central gateway provides stronger collaboration among ECUs, the ability to handle more complex functions (e.g. adaptive cruise control) and the potential of cross-functional connection.
- **Domain Centralized Architecture** combines domain controllers with a central gateway, enhancing the vehicle's wiring harness complexity. Each domain controller, functioning as a *high performance computing unit* (HPCU), manages domain-specific car components.
  - ▶ **Note 2.2.** HPCU is a multi-core ECU which, in particular, comprises a graphics processing unit (GPU), random access memory (RAM), and deep learning accelerators to process high computational power-demand tasks [24]. This scalable unit may integrate with Edge and Cloud computing and serves as the zonal gateway, centralizing vehicle functionality. Design and runtime configurations of HPCU are complex, addressing mixed-critical software aspects and diverse system requirements. The configuration process, often NP-hard, involves managing extensive safety and performance parameters, with updates posing potential risks and costs [25, 26].

This architecture varies in the number of interconnected ECUs and their respective ASILs. It faces increased complexity, especially in driving automation, due to a rise in sensor and actuator numbers, enhanced data processing needs, and greater demand for intelligent power distribution [27]. Domain-specific ECUs, linked to function-specific units, optimize architectural costs and function handling, such as in parking assistance systems [28, 23].

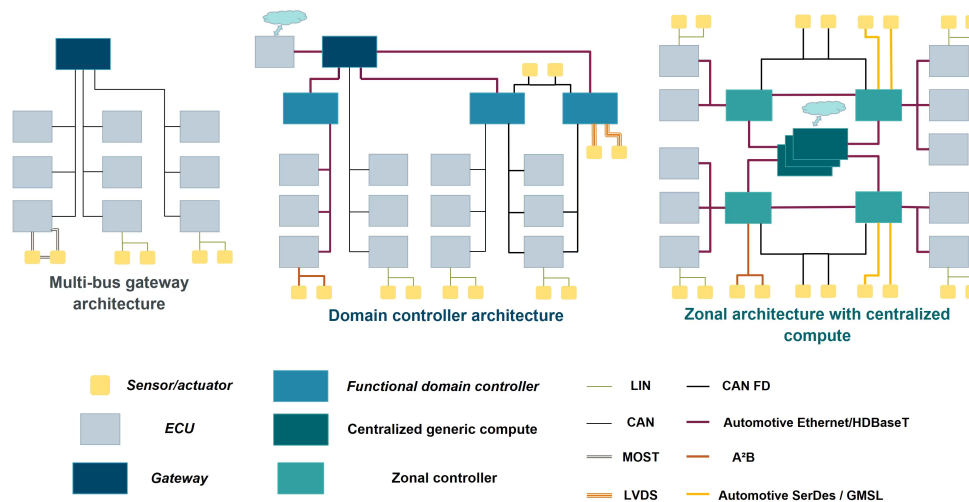
- **Zonal Architecture** comprises solely a *central* HPCU to manage complexity issues; it facilitates integration of future vehicle functions while achieving weight and cost savings [29, 22].

---

<sup>6</sup>Refer to section 2.3.1 for further details regarding CAN.



This architecture also includes a zonal ECUs, and function-specific ECUs. The HPCU, acting as a master and central gateway, processes data from various vehicle zones, interconnected via Automotive Ethernet (refer to section 2.3.5) for high-speed data transmission [30, 28]. Additionally, it supports virtual domains, enabling cloud-based function transfer and over the air (OTA) updates for the HPCU [31].



■ **Figure 2.4** An illustration of the main types of E/E architectural philosophies and technologies used for interconnection of ECUs [20].

Summarizing, E/E architecture of a car is primarily outlined by its supported features, since they could demand increased sensor integration, data processing & fusion, safety and especially a robust and high-speed communication network within the vehicle, among the other requirements. These requirements make car manufacturers opt for establishment of domain centralized or zonal architectures, which facilitate the integration of advanced features, like driving automation. Depending on the E/E architecture used in a particular car, the SWCs are implemented on different types of ECUs which are, in turn, interconnected using different communication buses.

## 2.3 Automotive Networking

SAE has classified the automotive networks into Classes A, B, and C with increasing order of criticality on real-time and dependability constraints [32, 33].

Class A, as per SAE's first classification, supports a data rate up to 10 kbps, suited for low-end, nonemission diagnostic, general-purpose communication, particularly in the *body & comfort* domain. These networks typically facilitate convenience features like actuators and smart sensors, controlling components such as lights, windshield wipers, doors, and seat adjustments, etc. Latency in Class A networks ranges from 50 to 150 ms [33].

Class B networks, supporting data rates between 10 kbps and 125 kbps [34], are primarily used for non-diagnostic, non-critical communications for general information transfer, such as instrumentation and emission data. These networks facilitate data transfer (e.g., parametric data values) between nodes and can reduce redundant sensors by supporting event-driven and periodic transmissions, including sleep/wakeup functions. Information shared over Class B networks is

not critical for system operation, allowing for a wider response window with variable response times based on the application. Interconnection of dissimilar systems is also allowed within these networks.

Class C networks support data rates from 125 kbps to 1 Mbps, catering to critical and real-time control systems such as engine, suspension, traction, brake, and transmission control. The response window in these networks is much narrower compared to Class B networks.

Networks with data rates over 1 Mbps, often referred to as Class D, though not formally categorized by SAE, are typically used in multimedia applications and crucial for hard real-time critical operations [35], due to their superior qualitative characteristics. Additionally, Classes B, C and D are not specifically linked to any domain since the actual fields of application of these networks may vary.

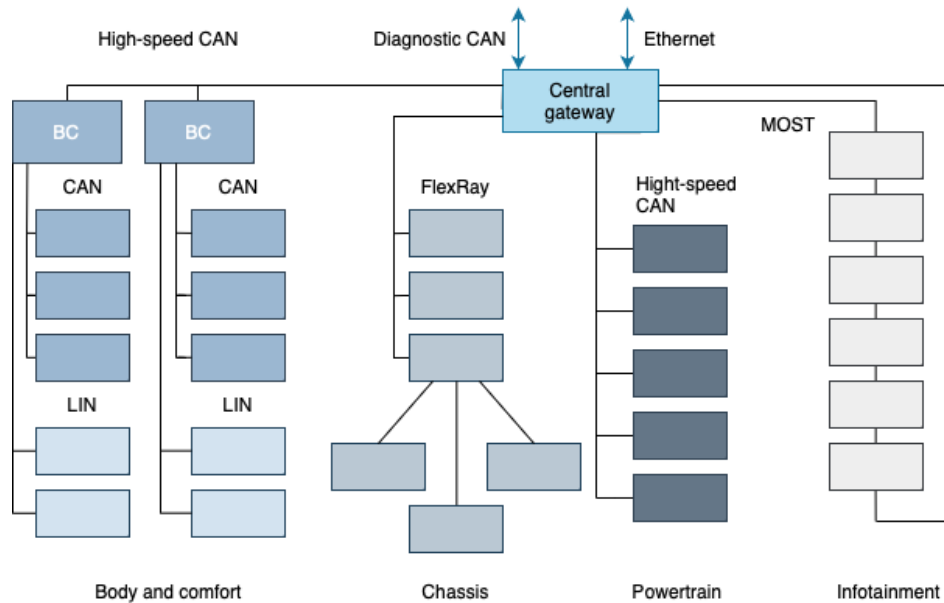
Table 2.4 presents a comparison of the main characteristics of various automotive protocols, further discussed in subsequent sections. It can be inferred that the attainment of higher levels of driving automation is infeasible without the deployment of Class D networks or equivalent technologies. In this context, Figure 2.5 schematically depicts an example of a distributed automotive network architecture showcasing the plausible common utilization of Class A-D communication protocols.

	Automotive Networking Protocols				
	<i>LIN</i>	<i>CAN</i>	<i>FlexRay</i>	<i>MOST</i>	<i>Ethernet</i>
<b>Classification</b>	Class A	Class B, C	Class D	Class D	Class D
<b>Application</b>	Body & comfort	Powertrain, driver assistance control (high speed); body and comfort low speed)	Chassis, driver assistance, safety control	Infotainment: stream data and control	Infotainment telematics, camera-based drivers assistance
<b>Topology</b>	Hierarchical bus	Hierarchical bus	Bus, star, multistar	Point-to-point	Star, point-to-point
<b>Media</b>	Single wire	Twisted-pair	Twisted pair or fiber	Optical	Twisted-pair
<b>Bit encoding</b>	NRZ	NRZ-5, MSb first	NRZ	BiPhase	Manchester Phase Encoding (MPE)
<b>Schedule approach</b>	Time	Event triggered	Time and event triggered	Event triggered	Event triggered
<b>Media access</b>	Master/slave	Contention	TDMA with priority	Master/slave	Contention
<b>Error detection</b>	8-bit CS	CRC	24-bit CRC	CRC	CRC
<b>Header length</b>	2 Bits/Byte	11 or 29 Bits	40 Bits	Not specified	14-22 Bytes
<b>Data length</b>	8 Bytes	0-8 Bytes	0-246 Bytes	Not specified	0-1500 Bytes
<b>In-message response</b>	No	No	No	No	Not specified
<b>Bit rate</b>	20 kbps	10 kbps-1Mbps	10 Mbps	25 Mbps	10 Mbps-100 Mbps
<b>Maximum bus length</b>	40 m	Not specified, typical 40m	Not specified	Not specified	100 m
<b>Maximum node</b>	16	Not specified, typical 32	Not specified	24	Theoretically 1024
<b>Cost</b>	Low	Medium	Medium	High	Not specified

■ **Table 2.4** Comparison of Automotive Protocols according to [34].

### 2.3.1 CAN

In 1991 the CAN bus (Controller Area Network) was the first bus system to be introduced to a motor vehicle in mass production [36]. It has since established itself as the standard system in the automotive sector, being also commonly used as a field bus in automation engineering in general.



■ **Figure 2.5** Example of distributed automotive network architecture.

The CAN bus is utilized in vehicle diagnostics and various domains within the motor vehicle. As a result of their different requirements, buses with different data rates are used that offer an optimum cost-benefit ratio for the field of application concerned. A distinction is made between CAN-FD, high-speed and low-speed CAN buses.

### 2.3.1.1 High-speed CAN (CAN-C)

CAN-C is defined in ISO 11898-2 standard and operates at bit rates of 125 kBit/s to 1 MBit/s. The data transfer is therefore able to meet the real-time requirements of the drivetrain. CAN-C buses are used for networking the following systems: *engine-management system, electronic transmission control, vehicle stabilization systems*.

### 2.3.1.2 Low-speed CAN (CAN-B)

CAN-B is defined in ISO 11898-3 standard and operates at a bit rate of 5 to 125 kBit/s. For many applications in the comfort/convenience and body area, this speed is sufficient to meet the real-time requirements demanded in this area. Examples of such applications are: *control of the air-conditioning system, seat adjustment, power-window unit, sliding-sunroof control, mirror adjuster, lighting system, control of the navigation system*.

### 2.3.1.3 CAN-FD

The primary difference between the classical CAN and CAN-FD is the Flexible Data (FD) supported by it. Using CAN-FD, electronic control units are enabled to dynamically switch between different data rates and longer or shorter messages [37]. Faster data speed and more data capacity enhancements results in several system operational advantages compared to classic CAN (according to ISO 11898-5, CAN-FD is specified for 2 and 5 MBit/s).

CAN-FD is typically used in high performance ECUs of modern vehicles.

### 2.3.2 LIN

The LIN bus was founded in 1998 and is suitable for low data rates of up to 20 kBit/s and is typically limited to a maximum of 16 bus subscribers [36].

► Note 2.3. The name, LIN (Local Interconnect Network), is derived from the fact that all electronic control units are located within a demarcated installation space (e.g. in the door). The LIN, therefore, is a local subsystem for supporting the vehicle network by means of superordinate CAN networks.

As far as the network nodes are concerned (being connected via LIN's electrical interface), a distinction is made between the master, which is generally an electronic control unit connected to a superordinate bus system, and the slaves. These are intelligent actuators, and switches with additional hardware for the LIN-bus interface. The bus subscribers are usually arranged in a linear bus topology and connected to each other by a single-wire line.

Communication on the LIN bus takes place in a time-synchronous manner, whereby the master defines the time grid. Consequently, there arises a strictly deterministic LIN bus response.

The LIN bus as a means of networking mechatronic systems can be used for many applications in the motor vehicle for which the bit rates and variability of the CAN bus are not essential. Featured examples of LIN applications include *control of the power-sunroof drive unit, of motors for seat adjustment, wiper motor for the windshield wiper, etc.*

### 2.3.3 FlexRay

FlexRay is a protocol that supports both time-triggered (primary) and event-triggered messaging (speed over 1 Mbps) in x-by-wire applications that need predictability and fault tolerance, as well as deterministic real-time and reliability communication. It is capable of a net data rate of 5 Mbps (2 channels, 10 Mbps gross) [34]. This protocol serves for safety-critical embedded systems and advanced control functions.

In FlexRay at the MAC level, a communication cycle merges a time-triggered (static) window and an event-triggered (dynamic) window. The time-triggered window employs a Time Division Multiple Access (TDMA) protocol for efficiency and determinism, while the event-triggered part uses flexible TDMA (FTDMA), dividing time into mini-slots for station transmissions. Communication cycles, executed periodically, prioritize higher-priority sources in the static part and lower priorities in the dynamic part. Nodes adapt to system configuration through messaging traffic. FlexRay's network topology is versatile, supporting bus, star, or multistars configurations, with optional channel redundancy.

### 2.3.4 MOST

MOST [38] is a multimedia fiber optic Class D network developed in 1998 by MOST Cooperation (a consortium composed of carmakers, set makers, system architects, and key component suppliers). The basic application blocks supported by MOST are audio and video transfer, based on which end-user applications like radios, GPS navigation, video displays and amplifiers, and entertainment systems can be built.

The MOST protocol defines data channels and control channels. The control channels are used to set up what data channels the senders and receivers use. Once the connection is established, data can flow continuously, delivering streaming data (Audio/Video). MOST provides point-to-point audio and video data transfer with a data rate of 24.8 Mbps [34].

## 2.3.5 Automotive Ethernet

The Ethernet, developed in the 1970s by Xerox Corporation, is a communication network technology used mainly in local area networks (LANs); compared to the other protocols, the use of the Ethernet in cars is a relatively new development [2]. Ethernet is based on the Open Systems Interconnection (OSI) model (refer to section 2.3.5.1) which groups similar communication functions together and provides defined interfaces between these groups. Its specifications are detailed in the IEEE 802.3 standard.

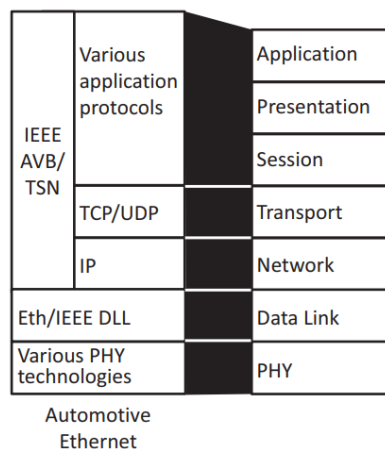
In comparison to the aforementioned networks, Automotive Ethernet provides higher bandwidth, higher security, and better fulfillment of the safety requirements based on ISO 26262 (refer to section 2.5.3). In addition, providing low latency in the communication network by using new message routing mechanisms plays a significant role in accelerating the progress of the in-car network for autonomous vehicles. Automotive Ethernet enables deterministic communication, ensuring that critical data is transmitted in a timely manner, making it suitable for *safety-critical applications in autonomous vehicles* such as powertrain, chassis, ADAS, infotainment systems, and body and comfort etc.

Several versions of the Ethernet exist beginning with the basic 10 Mbps version that uses a twisted-pair cable as a medium for a full-duplex and point-to-point communication (10BASE-T). Other versions are the 100BASE-T Ethernet, which is capable of a transmission rate of 100 Mbps, and the 1000BASE-T Ethernet, also called the Gigabit Ethernet. For vehicular network purposes, the 10BASE-T and 100BASE-T are the most popular as Broadcom's BroadR-Reach 10/100 PHY Ethernet module (which is currently the de facto standard for Ethernet in automotive application) supports these Ethernet versions.

Ethernet utilizes the CSMA/CD (Carrier Sense Multiple Access with Collision Detection) scheme to manage access to the transmission medium [39]. In essence, nodes within an Ethernet network contend with each other for the utilization of the transmission medium. When an Ethernet node intends to transmit a message, it first checks for any ongoing traffic on the shared transmission medium [2]. This process, known as carrier sensing, involves the node suspending transmission attempts if traffic is detected and waiting for the medium to become idle before making another attempt [39]. Conversely, if the medium is found to be free, the node proceeds to transmit its message.

### 2.3.5.1 ISO/OSI Model

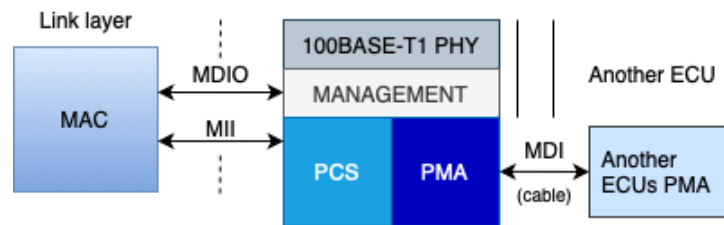
The ISO/OSI model of Automotive Ethernet, illustrated in Figure 2.6, delineates distinct layers and protocols participating in the communication process.



■ **Figure 2.6** ISO/OSI reference model and protocols used in context of Automotive Ethernet [2].

*Application* and *Presentation* layers are managed by an application responsible for dispatching data to the Automotive Ethernet. Within the *Session* layer, the structural arrangement of data within a packet is expounded. Subsequently, at the *Transport* layer, the data packet undergoes transformation into a TCP packet for reliable communication and a UDP packet if reliability is not paramount. Within the *Network* layer, the packet undergoes augmentation with pertinent information concerning the destination and source IP addresses, along with other network layer attributes. Below this stratum lies the *Data Link* layer, which appends the packet with Medium Access Control (MAC) addresses and other data inherent to the link layer. Ultimately, at the *Physical* layer, the packet is transmitted over 100/1000BASE-T1 Ethernet. An inverse sequence is employed during the reception of packets.

- 1. Physical Layer:** Key differences between Automotive Ethernet and usual one lie in the technologies used for data processing on the physical layer [2]. The physical layer delineates both physical and electrical attributes, encompassing key components such as Physical Medium Attachment (PMA) sublayer, the Physical Coding Sublayer (PCS), as well as elements like cables and connectors (refer to Figure 2.7).

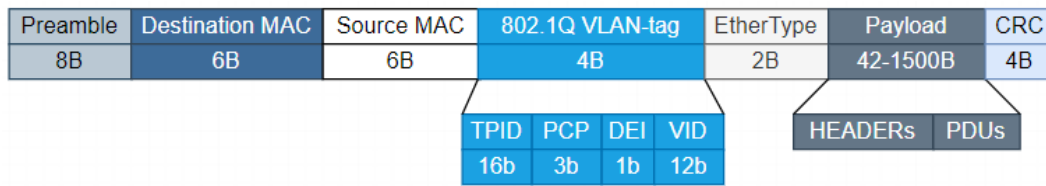


■ **Figure 2.7** 100BASE-T1 physical layers and its sublayers.

The PCS, situated within the physical layer (PHY), plays a pivotal role in data preparation between the Medium Access Control (MAC) sublayer of the Microcontroller Unit (MCU) and the PMA sublayer. Facilitating communication between these sublayers is the Media-Independent Interface (MII). Data from the MII undergoes several transformations, after which it is then serialized and introduced into the PMA sublayer. The PMA sublayer assumes responsibility for the control of data to/from the Medium-Dependent Interface (MDI). Management of the PHY is executed through the Media Data Input/Output (MDIO) from the MAC, which involves reading and writing to management registers via MDIO [2].

- 2. Data Link Layer:** The data link layer is responsible for forwarding data between adjacent nodes in the network, whether it be a Wide Area Network (WAN) or a Local Area Network (LAN). This layer incorporates the crucial element known as Media Access Control (MAC) [40].

The MAC, implemented in the network adapter of the Microcontroller Unit (MCU), is uniquely identified by the MAC address stored in the Electrically Erasable Programmable Read-Only Memory (EEPROM). Core functions of the MAC sublayer include regulating data flow, handling the transmission and reception of Ethernet frames, computing Cyclic Redundancy Check (CRC), and verifying the integrity of received frames. The structure of an Ethernet frame is depicted in Figure 2.8.



■ **Figure 2.8** Ethernet frame structure at data link layer.

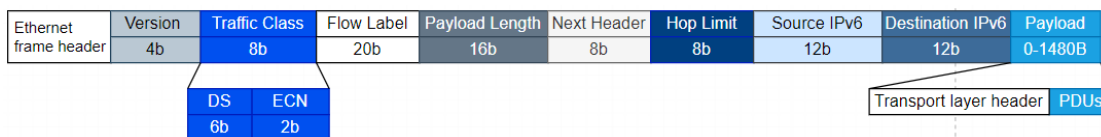
Within the structure of the data link layer packet presented, the initial segment comprises a Preamble 8B data field. The first 56 bits consist of alternating "1" and "0" bits, facilitating clock synchronization at the bit level among devices (7 bytes filled with 0xAA). The final byte of Preamble, known as the Start Frame Delimiter (SFD), serves the purpose of byte synchronization and signifies the commencement of a packet with a "1" bit (0xAB).

Following the preamble, the destination and source MAC addresses are outlined in MAC address fields. Subsequently, the 802.1Q tag (VLAN-tag) field follows, adhering to the IEEE 802.1Q networking standard that supports Virtual LANs (VLANs) on the Ethernet network. This field includes subfields such as VLAN Identifier (VID), Drop Eligible Indicator (DEI), Priority Code Point (PCP) and Tag Protocol Identifier (TPID) [41]. The TPID, positioned similarly to the EtherType field in untagged frames, is set to the value 0x8100 to identify the 802.1Q-tagged frame. The PCP field designates the frame's priority level, while DEI indicates frames that may be dropped during congestion. VID specifies the VLAN of the frame, with reserved values of 0x000 for frames lacking a VLAN ID and 0xFFF for implementation use.

Following the VLAN-tag field is the EtherType field, used to specify the size of bytes if the value is  $\leq 1500$ . If the value is  $\geq 1536$ , this field indicates the encapsulated protocol in the payload [42]. The length of the frame, in this case, is determined by the Interpacket Gap (IPG). Subsequent to the EtherType field is the Payload field, housing upper ISO/OSI headers and Protocol Data Units (PDU's), described in section 2.4.1.3. Ultimately, the Cyclic Redundancy Check (CRC) field is appended to detect corrupted data.

**3. Network Layer:** The Network layer holds the responsibility of directing packets to the target network (LAN), thus determining both the destination and source networks. As per the ISO/OSI model mentioned earlier, the IPv6 protocol is employed.

In the initial phase, the IPv6 packet undergoes encapsulation into an Ethernet frame packet (refer to Figure 2.9). The packet is signified to be IPv6, by the Version with a value of 0x6. Following this is the Traffic Class field, segmented into subfields Differentiated Services (DS) and Explicit Congestion Notification (ECN) [42]. The ECN denotes whether the source implements congestion control or not, while the DS is utilized for packet classification.



■ **Figure 2.9** IPv6 packet structure at the network layer.

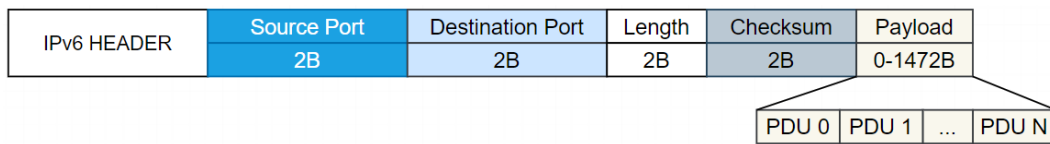
The Flow Label field serves as an identifier for a packet flow, representing a group of packets such as a media stream, between the source and destination. The subsequent field is Payload Length, defining the size of the Payload field in bytes. Next is the Next Header field, specifying the type of transport header. Following this is the Hop Limit field, which replaces the Time To Live field in IPv4 and is decremented with each forwarding. When the Hop Limit reaches a value of 0, the packet is discarded.



The subsequent Source Address and Destination Address fields describe the IPv6 addresses of the sending and receiving nodes. Eventually, the payload field is composed of the transport layer header, and the structure of Protocol Data Units (PDUs) (refer to section 2.4.1.3 for some details).

- 4. **Transport Layer:** The Transport layer assumes the responsibility of facilitating host-to-host communication and provides essential services such as flow control and multiplexing. Within the ISO/OSI model mentioned earlier, both the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are anticipated.

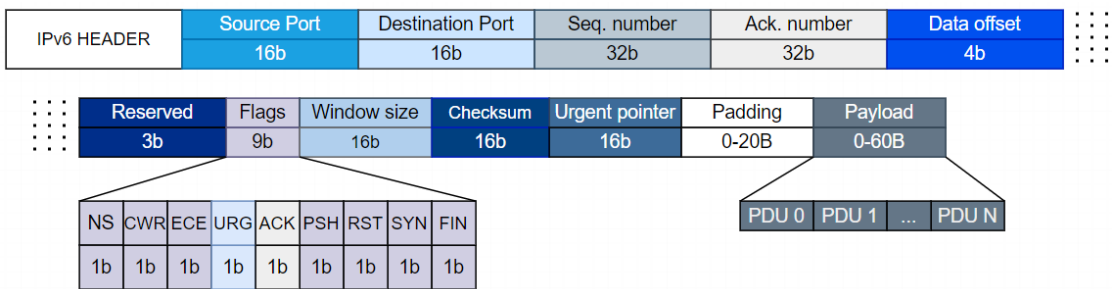
UDP, being a straightforward and unreliable communication protocol suitable for non-critical data transfer, structures its packet header with four fields (refer to Figure 2.10). The initial two fields define the Source and Destination ports, followed by the Length field, indicating the total size of the UDP packet. The fourth Checksum field is IPv6 specific and serves to detect errors in the header and data.



■ **Figure 2.10** UDP packet structure as defined in [43].

In contrast, TCP is a reliable and connection-oriented protocol, necessitating the establishment of communication. Consequently, the structure of a TCP packet is more intricate, as depicted in Figure 2.11.

Similar to UDP, the first two fields denote ports, followed by the Sequence number field providing details about the first data byte number. The Acknowledge number field follows, indicating the number of received data bytes. Subsequently, the Reserved field is designated for future purposes, while the Data offset field specifies the header's size in 32-bit words, ranging from a minimum of 5 to a maximum of 15 words [42].



■ **Figure 2.11** TCP packet structure.

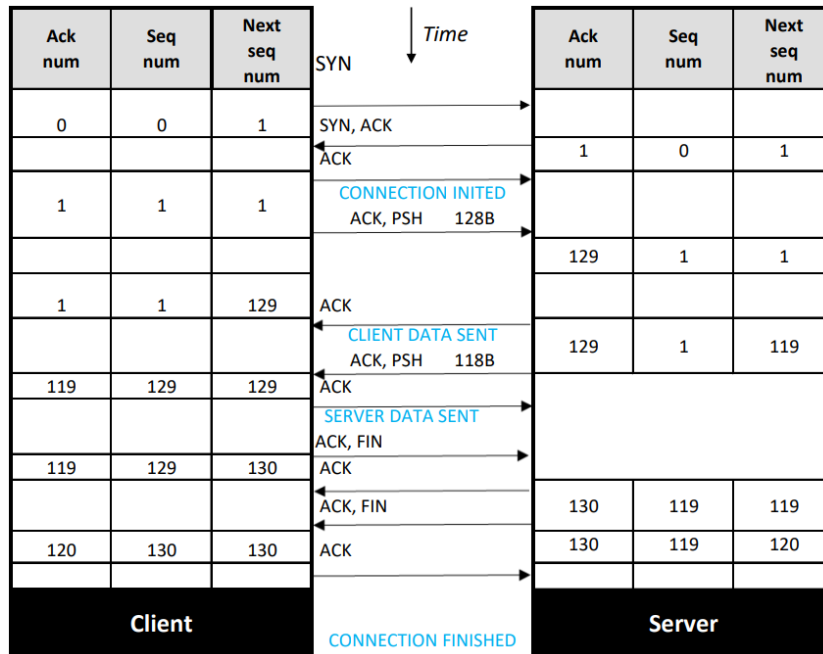
The Flags field plays a crucial role in communication, with notable flags including the Acknowledgement (ACK) flag signifying a valid Acknowledge number, the Push (PSH) flag set during data transmission, the Reset (RST) flag for connection reset, the Synchronize (SYN) flag for connection establishment, and the Finish (FIN) flag indicating the sender's last packet. Flags such as Urgent (URG), CWR, ECE, and NS are employed for Explicit Congestion Notification (ECN) and concealment protection [42].

The subsequent Window size field specifies the quantity of data bytes the sender can receive without sending an acknowledgment. The Checksum field serves for error-checking of the



header or payload. The Urgent pointer field, valid when the URG flag is set, denotes an offset from the start of the data field. The final field, Padding, ensures the header size is a multiple of 32, exceeding 20 bytes but remaining below 60 bytes.

In context of Automotive Ethernet, the payload of both TCP and UDP packets comprises PDUs and other protocol headers passed from the upper layers.



■ **Figure 2.12** Example of TCP communication.

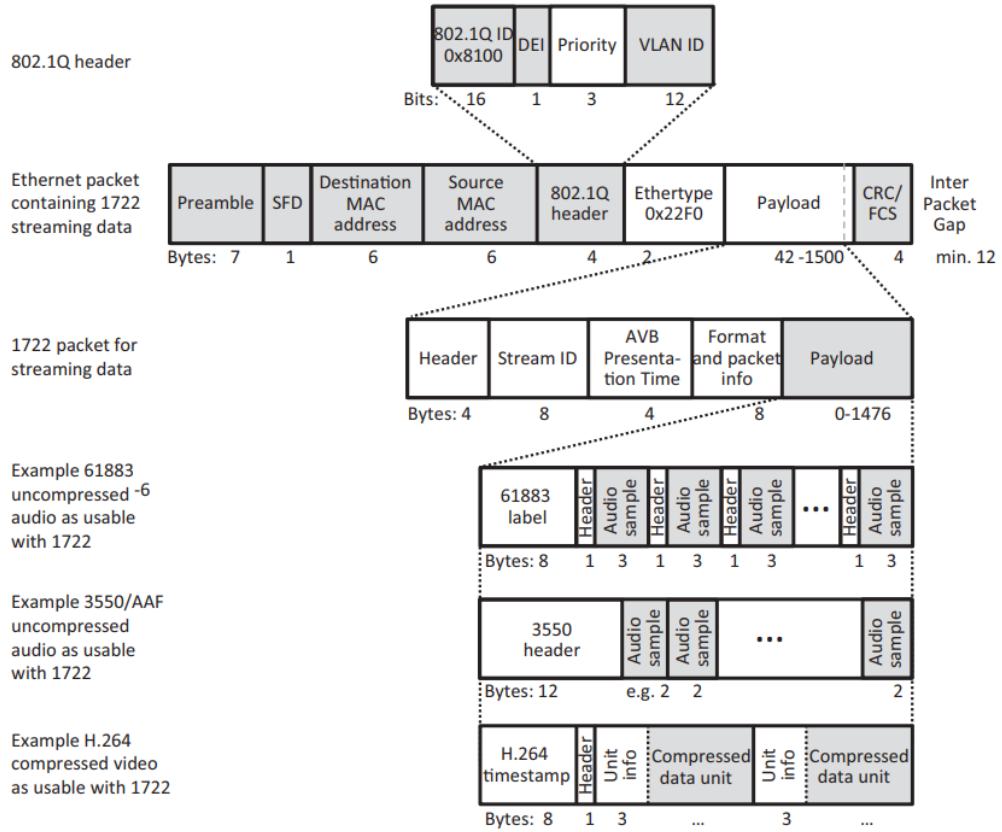
An illustrative example of TCP communication is presented in Figure 2.12, demonstrating a scenario where a server and client establish a connection, exchange data packets, and subsequently close the connection.

The establishment of a connection involves the transmission of a packet with a set SYN flag, initiating an internal sequence counter increment by the client. The server responds with an acknowledgment, featuring both ACK and SYN flags. Upon the client’s acknowledgment, the connection is established. During the established connection, both sides can send packets, with acknowledgments ensuring successful data transmission. To conclude the connection, one side sends a packet with a set FIN flag, prompting the other node to respond with a packet featuring a set FIN flag and awaiting acknowledgment.

- 5. **Session Layer:** Within the session layer of the ISO/OSI model, data is potentially structured as PDUs (see section 2.4.1.3), which may adhere to various protocols such as SOME/IP, ViWi, DoIP, or others used over the Automotive Ethernet and managed by AUTOSAR (refer to section 2.4.1.6).
- 6. **Presentation Layer:** The presentation layer is fully managed by AUTOSAR, refer to section 2.4.1.6.
- 7. **Application Layer:** The application layer is fully managed by AUTOSAR, refer to section 2.4.1.6.

► **Note 2.4.** Several exceptions apply to the Ethernet packet format when IEEE 1722 protocol is used in terms of Audio Video Bridging (AVB) or Time Sensitive Networking (TSN), refer

to Figure 2.13. Although it may be related to the ADAS/DAS sensors (transferring data from cameras, etc.) is not reviewed in details to reduce the scope of the thesis.



■ **Figure 2.13** The IEEE 1722 packet format with example 1722 payloads [2].

## 2.4 AUTOSAR

The architecture of automotive software systems, as software-intensive systems, can be seen from different views, such as *functional view*, *physical system view* and *logical view* [44]. The logical and the physical views deserve special attention in context of this thesis.

The logical architecture in automotive software is tasked with the design and organization of essential vehicle functions, such as implementing automatic braking upon detecting pedestrians in the vehicle’s path. These high-level functions are typically executed by several logical software components that interact through data exchange. Logical components are often organized into subsystems that align with the vehicle’s logical (E/E architectural) domains, based on the specific functions they perform.

On the other hand, the physical architecture in automotive software systems is characteristically spread across multiple Electronic Control Units (ECUs). These ECUs are charged with executing various high-level functions outlined in the logical architecture. This execution involves assigning logical software components, which are in charge of these functions, to specific ECUs. Consequently, these logical components are transformed into executable ECU application software components. Importantly, each logical software component is allocated to *at least one* ECU.

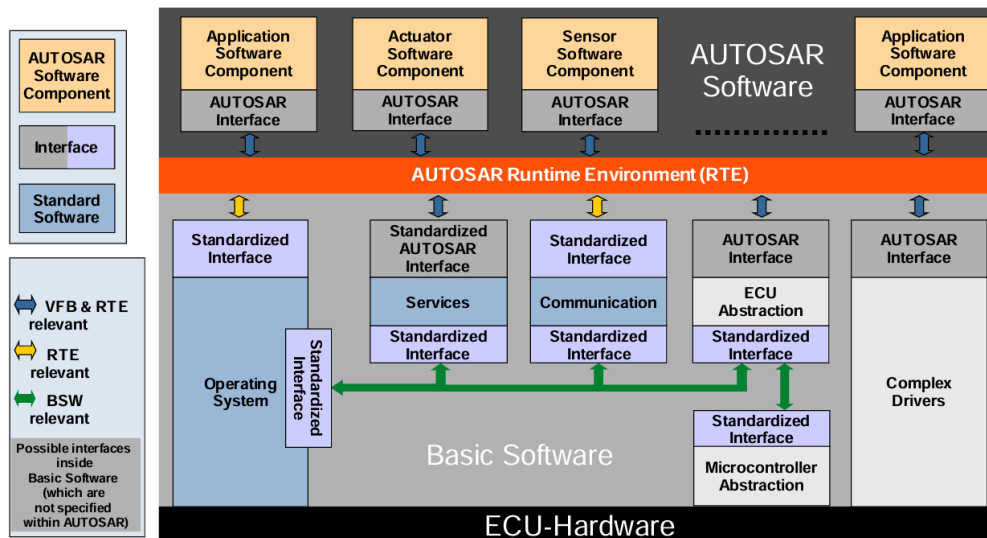
Apart from the physical system architecture that consists of a number of ECUs, each individual ECU possesses its distinct physical architecture. This architecture is primarily composed of the following key components [44]:

- **Application software** consists of a number of allocated software components and is responsible for executing vehicle functionalities realized by this ECU.
- **Middleware software** is responsible for providing services to the application software (e.g. transmission/reception of data on the electronic buses, and tracking diagnostic errors).
- **Hardware** includes a number of drivers responsible for controlling different hardware units (e.g. electronic buses and the CPU of the ECU).

The development of the logical and physical architectural views of automotive software systems and their ECUs is mostly done following the MDA (Model-Driven Architecture) approach [44].

To streamline the distributed creation and design of automotive software systems and their architectural elements, the AUTOSAR (AUTomotive Open Systems ARchitecture) initiative was launched in 2003. This initiative emerged from a collaborative effort among automobile manufacturers and their respective software and hardware suppliers. Presently, AUTOSAR has grown to encompass over 150 partners worldwide [45], thereby establishing itself as an industry standard within the automotive sector.

The design of ECU software following the AUTOSAR framework<sup>7</sup> adheres to a tri-layered architectural model. This model is established on top of the ECU's hardware layer, as depicted in Figure 2.14.



■ **Figure 2.14** AUTOSAR layered software architecture [45].

The first layer, *Application software*, consists of a number of software components that realize a set of vehicle functionalities by exchanging data using interfaces defined on these components. This layer is based on the logical architectural design of the system.

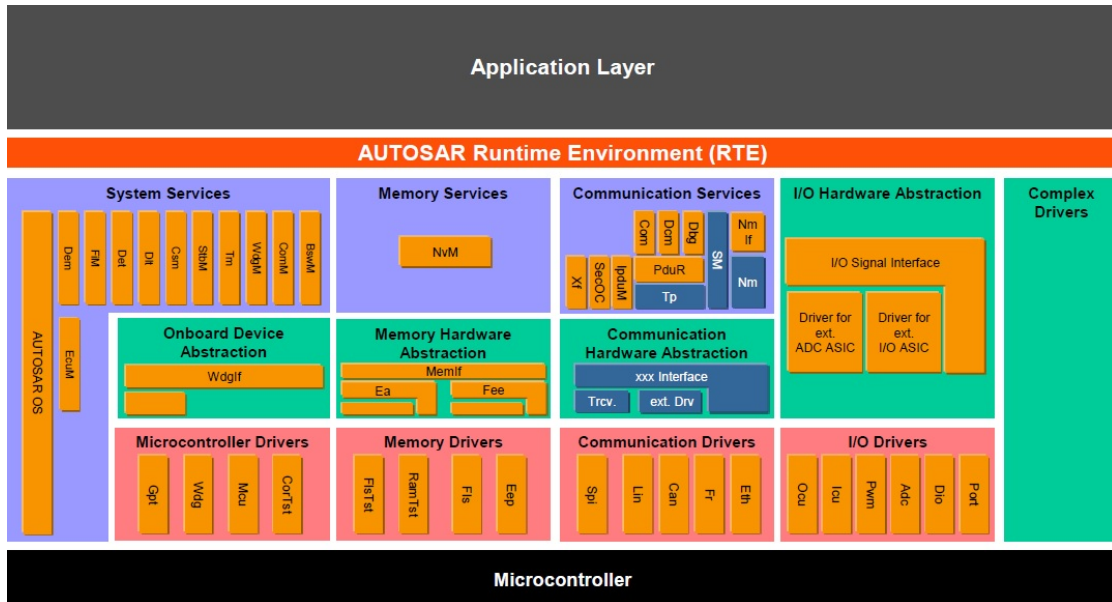
The second layer, *Run-time environment (RTE)*, controls the communication between software components, based on Virtual Functional Bus (VFB), abstracting the fact that they may be

<sup>7</sup>It should be noted that AUTOSAR encompasses both Classic and Adaptive platforms, differentiating in supported features.

allocated onto the same or different ECUs. If the software components are allocated to different ECUs, transmission of the respective signals on the electronic buses is needed, which is done by the *Basic software* (BSW) layer.

The Basic Software (BSW) encompasses various services, such as diagnostic protocols and memory management, within the automotive software framework. Additionally, it includes functionalities related to communication, I/O management, and network management, along with serving as an operating system. The BSW can be modularly divided into three distinct key layers: the *Services Layer*, *Hardware Abstraction Layer*, and *Microcontroller Abstraction Layer* (MCAL) [45].

The ECU abstraction, a part of the BSW, effectively decouples higher-level software components from the specific hardware dependencies of the underlying electronic control units, by providing a software interface that abstracts the electrical values and characteristics of each ECU. The MCAL plays a vital role in ensuring a standardized interface to the BSW modules, by managing microcontroller peripherals and providing microcontroller-independent values. As shown in Figure 2.15, these layers themselves can be further divided based on their specific purposes.



■ **Figure 2.15** Overview of AUTOSAR software layers & appropriate modules in detail [45].

## 2.4.1 Details on AUTOSAR Communication

To evaluate the potential for signal manipulation in alignment with predefined objectives (refer to section 1.2), it is essential to examine both the characteristics of the signals and the various phases involved in their transmission from the sender to the receiver. This analysis is crucial for understanding the processes of signal storage and transmission across the network.

### 2.4.1.1 Signal

In the context of AUTOSAR, a *signal* is equivalent to a message in the communication module according to OSEK COM standard [45]. In compliance with OSEK COM [46], a message defines a mechanism for data exchange between different entities and with other CPUs. An AUTOSAR signal represents a sequence of bits, which could be assigned to one of three types of possible

values (*initial*, *error* and *normal*) on the application level. These messages could be either of a simple data type or of a complex one.

Inside a complex data type, there are one or more data elements (primitive data types). RTE decomposes the complex data type in single signals and sends them (and/or signals that are of simple data type) to the AUTOSAR COM (Communication) module<sup>8</sup>. In this context, it's important to note that an AUTOSAR signal can be conveyed by one or more signals within the AUTOSAR COM.

### 2.4.1.2 Signal Group

A set of signals that must always be transmitted together in a common I-PDU<sup>9</sup> is defined as a *signal group*.

► Note 2.5. Determining which signals are grouped together into a specific signal group is considered an input for the COM generation process.

Signal groups ensure that AUTOSAR composite data types are transferred atomically, guaranteeing data consistency. A signal group has the following properties [47]:

1. A signal can belong to at most one signal group.
2. A signal group can not belong to more than exactly one I-PDU.
3. Signal groups do not overlap each other within an I-PDU
4. Signal groups are a contiguous set of signals which belong to this group, however it is possible to have unused bits ('holes') within a group.
5. Signal groups may contain no signals ('may be empty').
6. The signals that belong to a particular signal group are contiguous in the data stream.

Furthermore, a signal group (or an individual signal, respectively) can have several *transmission modes* assigned, describing the signal transmission within the network:

- **None:** This mode indicates that there is no transmission, meaning data is not actively sent.
- **Periodic:** transmissions occur indefinitely with a fixed period between them. This means that data is sent at regular intervals without any specific external trigger.
- **Direct/n-times:** event-driven transmission with  $n - 1$  repetitions. In this mode, data is transmitted in response to a specific event, and the transmission is repeated  $n - 1$  times.
- **Mixed:** this mode combines periodic transmission with direct/n-times transmissions in between. Data is sent periodically, and additional transmissions are triggered by events as needed.

Additionally, AUTOSAR COM supports *Update-bits*, which are mechanisms used to inform the receiver whether the sender has updated the data in a particular signal or signal group before sending it. However, these Update-bits are not allowed when using direct/n-times transmission mode with  $n > 1$  [47].

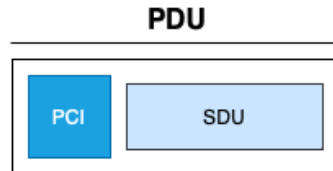
---

<sup>8</sup>Refer to Figure 2.19.

<sup>9</sup>Refer to 2.4.1.3 for details on naming conventions.

### 2.4.1.3 Protocol Data Units

Communication from AUTOSAR COM to lower modules and vice versa is done through PDUs (Protocol Data Units). Generally, a PDU is the basic generic data transfer unit for different vehicle network communication protocols (Ethernet, CAN, FlexRay, LIN, etc). Each PDU incorporates PCI (Protocol Control Information) and SDU (Service Data Unit), as shown at Figure 2.16.

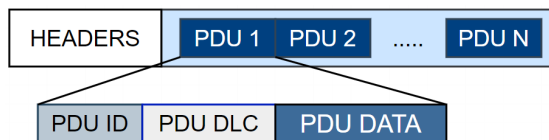


■ **Figure 2.16** Encapsulation of data (an SDU) by adding a header (the PCI) to form a Protocol Data Unit processed by a lower layer.

The PCI is added by a protocol layer on the transmission side and is removed on the receiving side; it contains RTE-related source and target information. This information is needed to pass SDU from one instance of a specific protocol layer to another instance. In context of AUTOSAR, PCI is identified as AUTOSAR PDU header, while SDU stands for payload of attached signals.

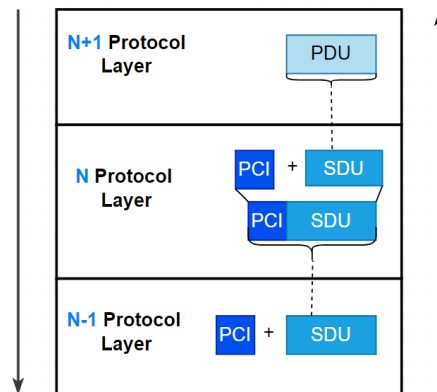
The AUTOSAR PDU header could be of different length [48], depending on the environment set: 32 bits (*short* headers), 64 bits (*long* headers). Importantly, as depicted on Figure 2.17, AUTOSAR PDU header includes the following fields:

- **ID** is assigned statically and is used to identify PDUs throughout communication. When short headers are applied, it takes 24 bits and 32 bits otherwise.
- **DLC** determines the overall length of a PDU. When short headers are applied, it takes 8 bits and 32 bits otherwise.



■ **Figure 2.17** Internal structure of Protocol Data Unit within the packet payload.

The signals coming from the different applications from the RTE layer are coupled together to the data segment of a PDU (SDU), which can be interpreted as low levels PDU which does not contain any header information. SDU is the abstraction for modules beneath PDUR (PDU Router, refer to Figure 2.19) whereas, for upper levels of PDUR, the PCI is excluded to process only the data structure (see Figure 2.18).



■ **Figure 2.18** Protocol Header Processing for Transmission (direction – down) & Reception (direction – up) by layer N.

Within the context of AUTOSAR communication, PDUs can contain layer-specific prefixes to distinguish them from each other. These PDUs are categorized into *I-PDUs* (Interaction Protocol Data Unit), *L-PDUs* (Data Link Layer Protocol Data Unit), and *N-PDUs* (Network Layer Protocol Data Unit).

- **I-PDU (Interaction Protocol Data Unit):** Utilized in data communication between modules over the PDURouter, predominantly involving AUTOSAR COM and AUTOSAR DCM. I-PDUs, integral to AUTOSAR COM, are composed of multiple signals and are grouped singularly within I-PDU groups. Their length is contingent on the L-PDU's maximal length from the underlying communication interface.
- **L-PDU (Data Link Layer Protocol Data Unit):** Functioning within the AUTOSAR Hardware Abstraction Layer, L-PDUs encompass the Identifier, Data Length Code (DLC), and data (L-SDU). Their length is subject to the specific communication type, such as LIN or Automotive Ethernet, aligning with the Communication Hardware and Microcontroller Abstraction Layers.
- **N-PDU (Network Layer Protocol Data Unit):** A network layer PDU, N-PDU is a key component in the AUTOSAR TP Layer.

#### 2.4.1.4 I-PDU Group

In AUTOSAR COM, an I-PDU group is an arbitrary collection of I-PDUs and can contain zero or more I-PDUs or other I-PDU groups of the same direction (send or receive). An I-PDU that is part of another I-PDU group cannot contain another I-PDU group, which limits the I-PDU group hierarchy to two levels. Furthermore, no I-PDU group can be included in more than one other I-PDU group, and an I-PDU group must not contain itself.

It's important to note that a combination of received I-PDUs and sent I-PDUs within a single I-PDU group is not allowed [48]. By default, I-PDU groups in AUTOSAR COM are stopped, and routines are provided to start and stop them as needed, and total number of I-PDU groups is limited to a predefined value (usually 32).

#### 2.4.1.5 I-PDU Multiplexing

In order to optimize data transmission over the network via different protocols (e.g. Ethernet, CAN) several I-PDUs (also those, within a group) can be multiplexed into one resulting PDU that is transferred on the bus. Two approaches exist to accomplish it:

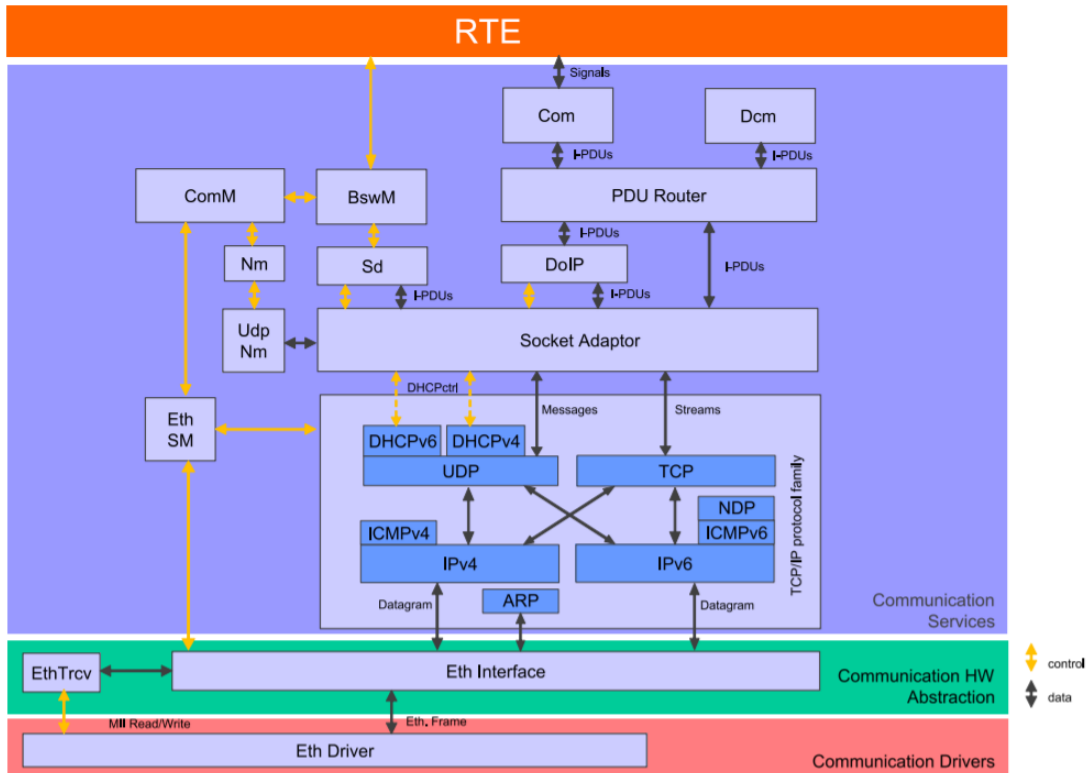
- **I-PDU Multiplexing:** this approach involves using the same I-PDU ID transferred from the PDU Router to the Communication Hardware Abstraction Layer with more than one unique layout of its SDU. It is implemented via introducing a *selector field* as a piece of the SDU of the multiplexed PDU. It is used to distinguish the contents of the multiplexed PDUs from each other.
- **Multiple PDU to Container Mapping:** this method entails collecting several I-PDUs into one Container PDU. This Container PDU is then transferred via PduR as one large I-PDU. This approach leverages the advantage of larger frame sizes in newer bus systems (Ethernet), allowing for efficient use of bandwidth in combination with smaller I-PDU sizes, typically 8 bytes.

The resulted PDUs could have either *static* or *dynamic* layout, meaning that their size can be either fixed or determined in the run-time depending on the general configuration of a system. For transmission of container PDUs with a static layout, minimum delay time cannot be ensured if two or more contained PDUs have the same MDT configuration [48].

On sender-side, the I-PDU Multiplexer module is responsible to combine appropriate I-PDUs from COM to new, multiplexed I-PDUs and send them back to the PDU Router. On receiver-side, it is responsible to interpret the content of multiplexed I-PDUs and provide COM with its appropriate separated I-PDUs, taking into account the value of the selector field.

#### 2.4.1.6 I-PDU Transmission

AUTOSAR manages the upper layers of the ISO/OSI model described at section 2.3.5.1, by using SOME/IP protocol [49]. It passes all the signals in form of PDUs to lower layers of ISO/OSI model via AUTOSAR PDU Router (see Figure 2.19).



■ **Figure 2.19** Modules of possible AUTOSAR Ethernet stack [50].

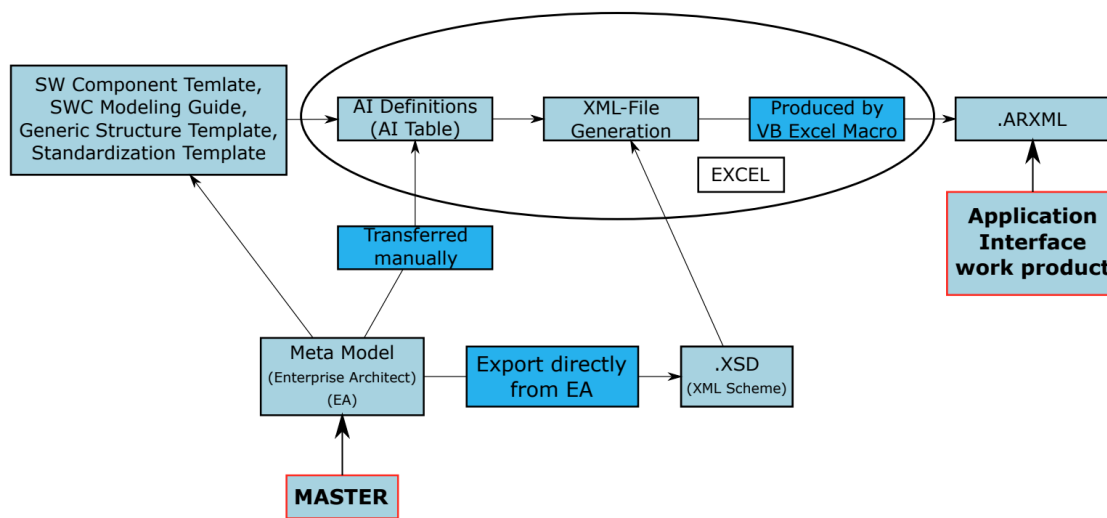


PDU Router manages the data received from AUTOSAR COM, AUTOSAR I-PDU Multiplexer etc. or other lower modules via *Routing Table*, transmitting the obtained I-PDUs and/or I-PDU groups in necessary direction. If the target SWC that is to receive the data is implemented in separate ECU, the received I-PDUs are passed to corresponding interfaces responsible for sending & receiving the data via physical medium (e.g. Automotive Ethernet, CAN, etc.), becoming N-PDUs and L-PDUs, consequently.

In context of Automotive Ethernet, L-PDUs transmitted over the physical medium directly represent the TCP/UDP packet payload.

## 2.4.2 AUTOSAR XML

ARXML (AUTOSAR XML) emerges as a pivotal component derived from the extensive AUTOSAR UML2.0 meta-model [51], as visually represented in Figure 2.20.



■ **Figure 2.20** The procedure of ARXML creation according to the AUTOSAR meta-model.

This meta-model provides a detailed specification for AUTOSAR systems. Within this framework, the `.ARXML`<sup>10</sup> file plays a crucial role, storing essential data related to application interfaces.

The AUTOSAR systems specification encompasses an extensive volume of data, with each subsystem receiving a detailed description. For instance, in the context of Automotive Ethernet, the specification for Ethernet frame encapsulation includes crucial details such as frame type, frame length, Ethernet addresses (MAC addresses), payload content, signals definitions (meaning, start bit, start byte, transmission mode, value, etc.) & PDU and multiplexing specifications, signal-to-PDU mappings, and other essential parameters at each level of internal processing. To maintain a manageable data structure, AUTOSAR employs a thematic organization, categorizing information into logical groups and subgroups, which are interlinked through relative path references [52]. Different `.arxml` files exist in the scope of AUTOSAR, each aiming at specific goals.

An example, outlining the selected fragment of plausible ARXML inner structure is provided at Listing 2.1.

■ **Code listing 2.1** Fragment of plausible ARXML inner structure.

```

<AR-PACKAGE >
  <SHORT-NAME>PDU</SHORT-NAME >
  
```

<sup>10</sup>Hereinafter, the `.extension` construct shall be used for denoting the file(s) of specified extension.

```

<ELEMENTS >
  <I-SIGNAL-I-PDU >
    <I-SIGNAL-TO-PDU-MAPPINGS >
      <I-SIGNAL-TO-I-PDU-MAPPING >
        <SHORT-NAME >Signal1 </SHORT-NAME >
        <I-SIGNAL-REF DEST="I-SIGNAL" >/ISignal/S1 </I-SIGNAL-REF >
        <PACKING-BYTE-ORDER >MSB-LAST </PACKING-BYTE-ORDER >
        <START-POSITION >0 </START-POSITION >
        <TRANSFER-PROPERTY >PENDING </TRANSFER-PROPERTY >
      </I-SIGNAL-TO-I-PDU-MAPPING >
    </I-SIGNAL-TO-PDU-MAPPINGS >
    <UNUSED-BIT-PATTERN >0 </UNUSED-BIT-PATTERN >
  </I-SIGNAL-I-PDU >
</ELEMENTS >
</AR-PACKAGE >
<AR-PACKAGE >
  <SHORT-NAME >ISignal </SHORT-NAME >
  <ELEMENTS >
    <I-SIGNAL >
      <SHORT-NAME >S1 </SHORT-NAME >
      <INIT-VALUE >
        <NUMERICAL-VALUE ><VALUE >128 </VALUE ></NUMERICAL-VALUE >
      </INIT-VALUE >
      <LENGTH >5 </LENGTH >
    </I-SIGNAL >
  </ELEMENTS >
</AR-PACKAGE >

```

ARXML proves to be a valuable tool for efficiently interpreting data packets, thanks to its comprehensive descriptions. Generally, XML format makes data accessible without the need for specialized software and ensures human readability. However, it's worth noting that this approach may introduce computational complexity due to the relatively low ratio of useful data and values in comparison to less useful elements like node names, tags and special characters.

## 2.5 Security of in-vehicle communication

The protection of in-vehicle communication is complicated and adapted to the circumstances found in a car (e.g. long product lifecycle, limited resources). It is generally achievable by ensuring *Physical security*, *Network security*, *ECU hardening* and *Application security* of all communication parties [2]. Among all these measures, aimed at safeguarding of in-vehicle communication, solely the Network security is to be analyzed to align with the scope of the thesis.

The primary objectives of *Network security* encompass ensuring *Confidentiality*, *Integrity*, and *Availability*, collectively referred to as *CIA*<sup>11</sup>. Network security's objective is to prevent unauthorized access to confidential data, guarantee data remains unaltered and originates from a verified source (upholding integrity), and ensure the consistent availability of services as intended.

Within a vehicle, potential attack targets include any ECU or communication line. Consequently, each requires individual protection against diverse generic attack categories [2]:

1. **Reading communication** (*confidentiality*).
2. **Replaying, changing, or injecting communication** (*integrity*).

<sup>11</sup>The exact origins of the 'CIA Triad' expression appear to be unknown, but the underlying concepts were already operative in military contexts millennia ago, as can be seen in the works of the *consulus*, *pontifex maximus* Gaius Julius Caesar[53].

- 3. **Selectively removing communication** (*integrity, availability*).
- 4. **Denial of Service** (*availability*).
- 5. **Attacking vulnerabilities of ECUs** (*confidentiality, integrity, availability*).

It is clear that the successful execution of attacks within categories 1-3 intersects with the thesis’ problematics and depend on the access the attacker has. In other words, the goals set in section 1.2 could mean to successfully perform emphasized attacks on in-vehicle Automotive Ethernet communication (with full access presumed). Consequently, an in-depth evaluation of security solutions aimed at thwarting such attacks becomes essential.

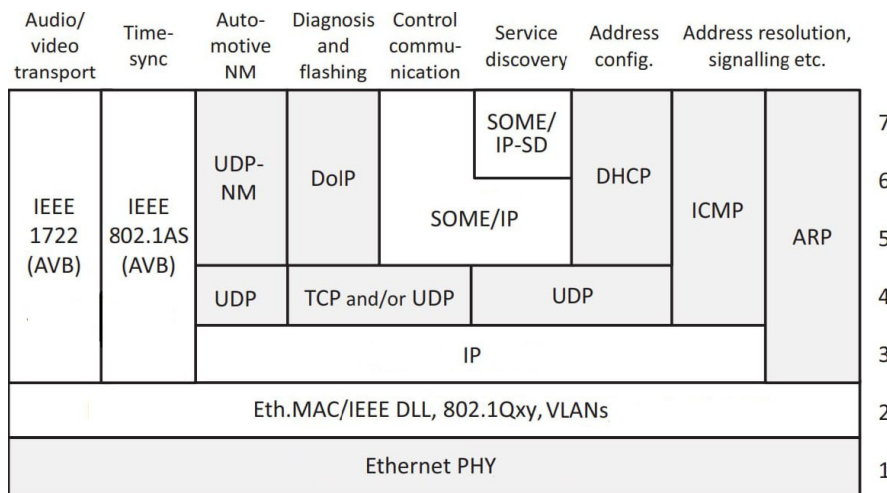
Given the multitude of potential attacks, a single security mechanism is insufficient for comprehensive communication protection. Consequently, automotive security systems typically employ a layered approach, integrating a combination of various mechanisms.

At the time of writing, the automotive industry has not reached a consensus on specific algorithms and protocols for Automotive Ethernet security, nor has there been an industry-wide standardization effort. Several distinct initiatives have surfaced, each targeting different aspects of this issue [2]: AUTOSAR SecOC, SAE J3061, ISO 21434, JASPAR, etc.

Broadly, the security of automotive networking can be considered from three perspectives. Since the Automotive Ethernet relies on the ISO/OSI model, the first perspective is the security mechanisms usually used in it, while the second is automotive-specific security additions integrated into the inter-ECU networking. The third perspective addresses the security concerns in the context of functional safety.

### 2.5.1 Security in context of ISO-OSI Model

Since the automotive Ethernet communication protocols are mainly structured in the ISO-OSI layer model (Figure 2.21), the same can be done for the network security solutions.



■ **Figure 2.21** Protocol overview for Automotive Ethernet [2].

Each security solution is specific to a layer, protecting it and the higher layers. Figure 2.22 illustrates this concept, showing security solutions across different layers of the Ethernet-based communication stack. The following list provides brief descriptions of these solutions as applied in the ISO/OSI model, typically used in Ethernet communication:

- **MACsec**, standardized as IEEE 802.1AE, secures layer two of the Ethernet stack, providing point-to-point (P2P) encryption and authentication for each connection [54]. This process,

encompassing VLAN tag protection, requires Ethernet switches to update authentication and encryption at each switch transition. MACsec's authentication algorithm is notably relevant to the automotive industry for ensuring integrity across mixed security domains. However, its implementation in vehicles necessitates hardware support in controllers and switches, increasing semiconductor costs [2]. Its principal benefit is the comprehensive protection of all communication types—unicast, multicast, and broadcast—across protocols, thereby significantly reducing the attack surface against external threats.

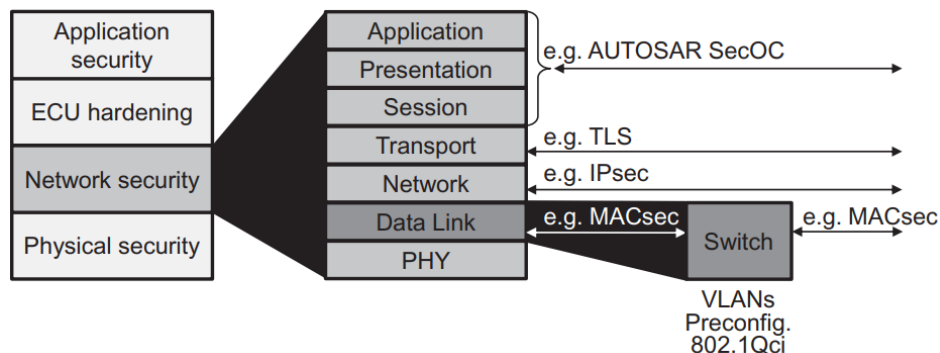
- **IPsec**, or Internet Protocol Security, was developed to supplement IP at layer three, addressing two main use cases. Firstly, it ensures end-to-end privacy, authenticity, and integrity, commonly applied within enterprise networks for server communication. Secondly, it facilitates both site-to-site and end-to-site Virtual Private Networks (VPNs). Using mechanisms such as encryption and an added header for message authentication, IPsec integrates directly at layer three of the ISO/OSI model, making it transparent to higher-layer applications [55]. Originally developed alongside IPv6, it is also compatible with IPv4.

IPsec's Authentication Header (AH) variant offers authentication without encryption, allowing non-IPsec systems to process packets by skipping the AH header. While mainly for IP-based communication, IPsec's layer three implementation enables protection of numerous protocols, excluding VLAN-Tags, (g)PTP, Ethernet, ARP, and NDP. Although multicast protection was not initially part of IPsec, experimental extensions now exist for this purpose [2].

- **TLS** or Transport Layer Security, formerly SSL, ensures privacy and data integrity for applications like HTTP, IMAP, SMTP over TCP by offering encryption and authentication [56]. It supports various symmetric/asymmetric encryption, key exchange, and authentication methods. TLS 1.2, introduced in 2008, was succeeded by the more secure and simplified TLS 1.3 in RFC 8446 (2018) [56], which resembles IPsec in its authentication-encryption approach.

For UDP, the DTLS (Datagram TLS) variant exists [57]. TLS and DTLS safeguard TCP- or UDP-based protocols (e.g., SOME/IP, HTTP) but do not extend protection to TCP, UDP, IP, VLAN-Tags, (g)PTP, Ethernet, or helper protocols such as ARP and NDP. They also exclusively protect unicast communications.

- **gPTP** or Generic Precision Time Protocol, is a time synchronization standard within time-sensitive networking (TSN), derived from the IEEE 1588 Precision Time Protocol (PTP). The protocol enables sub-microsecond levels of synchronization [2], crucial for networked ECUs in high ASIL ADAS functionalities. It functions through the exchange of timestamped messages between nodes, establishing a precise time reference. This protocol is adaptable for hardware or software implementation and is not restricted to any specific vendor.



■ **Figure 2.22** Layered automotive security approach and related mechanisms [2].

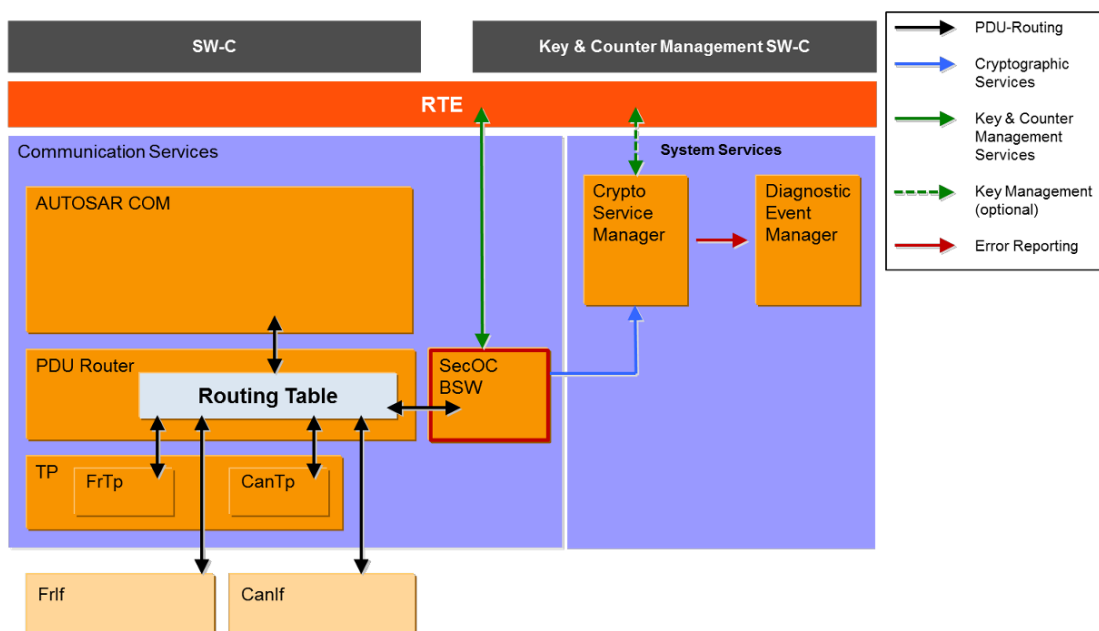
The exact usage of denoted mechanisms depends on the ECU resources and its intended functionality, leaving some of these solutions simply not applicable for the general in-vehicle communication. Summarizing, these security mechanisms or their subvariants (possibly developed by the car manufacturer) have to be individually chosen, set and configured, with the further factors considered, making their usage non-uniform and situation-dependent.

Another security solution which is closely related to the topic is CRC protection of an Ethernet frame (see Figure 2.8) itself, described in section 2.3.5.1. The frame check sequence (FCS) is a four-octet cyclic redundancy check (CRC) that allows detection of corrupted data within the entire frame as received on the receiver side. According to the standard [42], the FCS value is computed as a function of the protected MAC frame fields: source and destination address, length/type field, MAC client data and padding (that is, all fields except the FCS).

Per the standard, this computation is done using the left shifting IEEE 802.3 CRC-32 (polynomial = 0x4C11DB7, initial CRC = 0xFFFFFFFF, CRC is post complemented, verify value = 0x38FB2284) algorithm. The standard states that data is transmitted least significant bit (bit 0) first, while the FCS is transmitted most significant bit (bit 31) first [42]. In compliance with it, receiver should calculate a new FCS as data is received and then compare the received FCS with the FCS the receiver has calculated in order to verify the authenticity and integrity of the data received.

### 2.5.2 Security in context of AUTOSAR

AUTOSAR Secure Onboard Communication (AUTOSAR SecOC) has been developed in order to provide a resource-efficient and practical security mechanism that seamlessly integrates into the AUTOSAR communication (Figure 2.23) and that, being at AUTOSAR level, can be used with all networking technologies supported by AUTOSAR (CAN (FD), FlexRay, Ethernet, LIN). It provides end-to-end authentication and integrity based on *message authentication codes* (MAC) and freshness values (counters or timestamps).



■ **Figure 2.23** Integration of the SecOC BSW with CAN.<sup>12</sup>

<sup>12</sup>Source: [https://www.linkedin.com/pulse/secoc-shalini-krishnakumar-sjuj/?trk=public\\_post](https://www.linkedin.com/pulse/secoc-shalini-krishnakumar-sjuj/?trk=public_post)

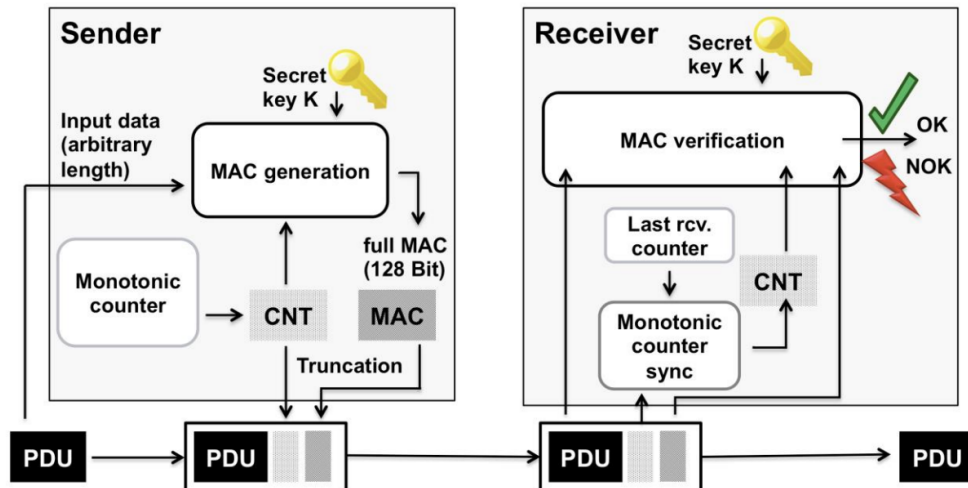
For efficiency in computation and bandwidth consumption it assumes symmetric keys, though neither asymmetric keys nor encryption are precluded [58]. SecOC has the unique capability to protect messages being passed, e.g., from CAN to Ethernet. AUTOSAR currently defines a mere core feature framework, leaving, for example, the key exchange, session establishment, and a concrete mechanism for freshness up to the manufacturer.

The nuances of AUTOSAR framework arise here that must be taken into account; AUTOSAR SecOC is not a mandatory component of a software architecture. The overall establishment of AUTOSAR SecOC requires both the sending ECU and the receiving ECU to implement a SecOC module.

On the sender side, the SecOC module creates a Secured I-PDU by adding authentication information to the outgoing Authentic I-PDU (whether multiplexed or not), as seen on the Figure 2.26. In practice this may be achieved by appending the authentication information to I-PDU in form of a signal(s) or by other means determined by a manufacturer.

On the receiver side, the SecOC module checks the freshness and authenticity of the Authentic I-PDU (regardless if the Freshness Value is or is not included in the Secure I-PDU payload) by verifying the authentication information that has been appended by the sending side SecOC module.

Figure 2.24 represents the principle of AUTOSAR SecOC functionality.

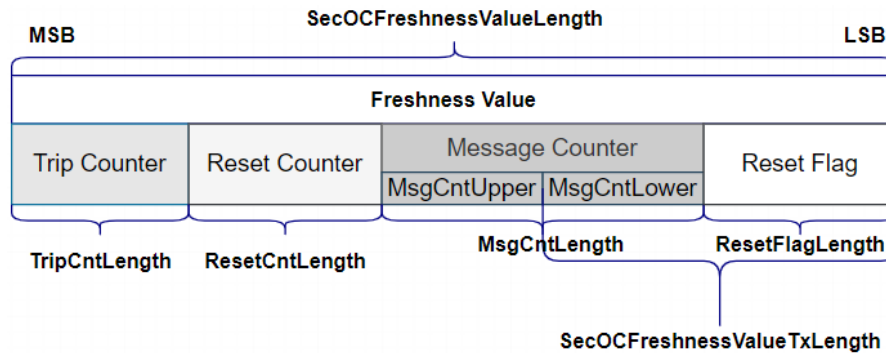


■ **Figure 2.24** AUTOSAR SecOC message authentication and freshness value verification [58].

Both MAC and freshness value counters can be additionally truncated, benefiting the size of transmitted data but decreasing the protection level. According to the AUTOSAR SecOC specification, the exact sizes of those vary depending on the AUTOSAR SecOC profile used:

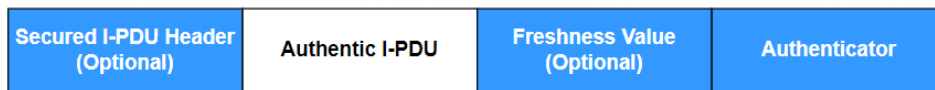
- **SecOC Profile 1 (or 24Bit-CMAC-8Bit-FV):** using the CMAC algorithm based on AES-128 according to NIST SP 800-38B to calculate the MAC, use the eight least significant bit of the freshness value (refer to Figure 2.25) as truncated freshness value and use the 24 most significant bits of the MAC as truncated MAC.
- **SecOC Profile 2 (or 24Bit-CMAC-No-FV):** using the CMAC algorithm based on AES-128 according to NIST SP 800-38B to calculate the MAC, don't use any freshness value at all and use the 24 most significant bits of the MAC as truncated MAC. The profile shall only be used if no synchronized freshness value is established. There is no restriction to a special bus.
- **SecOC Profile 3 (or JASPAR):** this profile shall be used for CAN and it depicts one configuration and usage of the JasPar counter base FV with Master-Slave Synchronization

method. It uses the CMAC algorithm based on AES-128 according to NIST SP 800-38B. Use the 4 least significant bits of the freshness value as truncated freshness value, and use the 28 most significant bits of the MAC as truncated MAC.



■ **Figure 2.25** Freshness Value structure.

Moreover, the precise configuration of AUTOSAR SecOC Profile applied to a particular in-vehicle communication environment in practice could differ from those, described in AUTOSAR SecOC specification. Manufacturer determines the algorithms used with selected I-PDUs (therefore, different algorithms could apply for neighbouring PDUs in the same packet), as well as sizes of both MAC and FV.



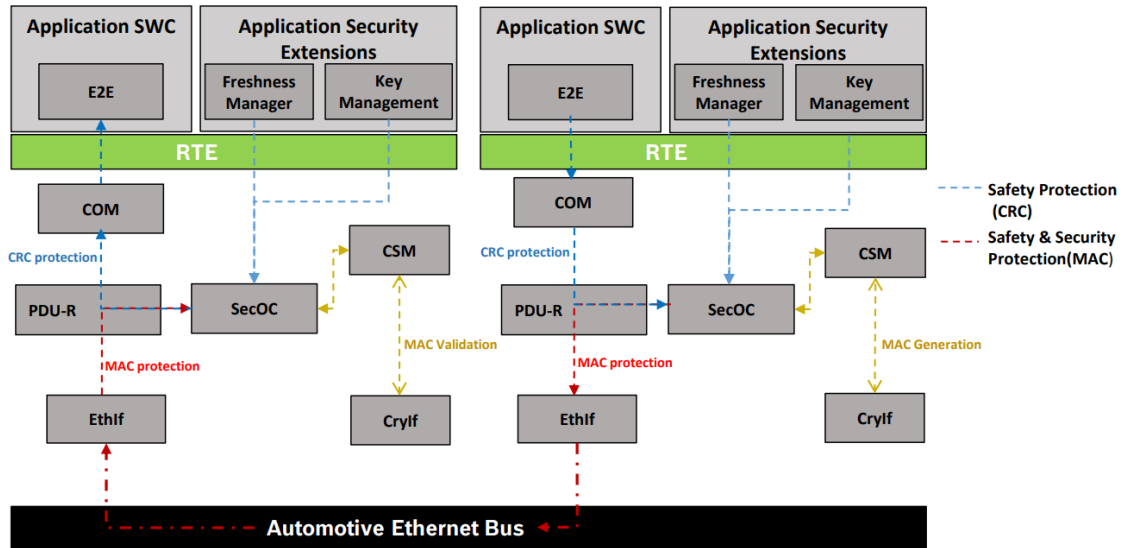
■ **Figure 2.26** Secured I-PDU structure.

### 2.5.3 Security in context of Functional Safety

Adhering to ISO 26262 necessitates additional measures for functional safety, focusing on 'end-to-end safety'. This principle requires that applications (SWC on an ECU) achieve specified safety targets. Consequently, applications must incorporate functions that can handle potential errors in the communication link or protocol stack, such as middleware. Two prevalent mechanisms for this are *application-based* CRC (Cyclic Redundancy Check) protection and the Alive Mechanism [14]:

1. In **application-based CRC protection**, a CRC is generated within the application context for specified data and sent alongside it. The receiver then verifies this CRC at the application level. Any communication errors from other layers will be reflected in the application-based CRC, obviating the need for additional reliability measures in those layers.
2. The **Alive Mechanism** aims to ensure data recipients can detect anomalies in the timing of the data transmission. Issues like delays or cessation in the transmitting application are critical. It's vital for the recipient to identify if outdated data is being cyclically transmitted instead of new information, especially as this poses a safety risk in various driving scenarios. To mitigate this, an alive counter or timestamp, originating from the application context, is transmitted, enabling the recipient to ascertain the operational status of the transmitter.





■ **Figure 2.27** Security mechanisms applied during the signal transmission from receiver's (to the left) and sender's (to the right) perspective [59].

Figure 2.27 represents the usage of both application-based CRC and AUTOSAR SecOC in context of AUTOSAR layered architecture.

The application of these mechanisms depends on the corresponding ASIL assignment, making those mechanisms typically being used in ECUs of ASIL B and higher, and therefore non-mandatory for ECUs with QM or ASIL A assigned [2]. In context of AUTOSAR, the exact values of both application-based CRC and alive counter (timestamp) are treated as signals in the PDU payload. These measures are not mandatory independent from each other, making possible a case when computation of application-based CRC depends on the alive counter or timestamp.

## 2.6 Existing Solutions

One of the prominent tools in domain of signals manipulation on Automotive Ethernet is the FlexDevice family [60], being a suite of multifunctional bus control units that provide comprehensive solutions for automotive networking.

These devices support a variety of bus systems and have several variable interfaces to cater to a wide range of applications. Importantly, the FL3X Switch 1000BASE-T1 is a notable product in this category, recognized as one of the first Automotive Ethernet Switches capable of operating at 1000BASE-T1 [61]. The primary features of the FlexDevice family (refer to Figure 2.28) relevant to signal manipulation in Automotive Ethernet are:

- **Gateway Functions:** The ability to act as gateways between different automotive bus systems.
- **Bus Interfaces:** Offering multiple bus interfaces for seamless integration and communication across various network types.
- **Remaining Bus Simulation (RBS):** Essential for testing environments where simulating bus traffic is required.
- **Signals & PDU Manipulation:** Direct control over signal and PDU parameters for development and testing.



- **Data Logging:** Capturing and analyzing data traffic within the network.
- **Rapid Prototyping:** Facilitating quick development and testing of new automotive network configurations.
- **Visualization:** Presents a graphical interface for monitoring and analyzing signal behavior within the network.
- **Gateway:** Provides the necessary tools for creating gateways between various bus systems.



■ **Figure 2.28** The multifunctional bus control unit FlexDevice-L.

In particular, the tools belonging to FlexDevice family support direct interaction with gPTP and AUTOSAR SecOC. Additionally, they support a broad spectrum of interfaces and bus systems, including [62, 63]:

- FlexRay controllers and channels.
- CAN-FD / LIN.
- Standard Ethernet (100BASE-TX) and Automotive Ethernet (1000BASE-T1, 100BASE-T1).

It is crucial to acknowledge that a device belonging to FlexDevice family and compatible with Automotive Ethernet, was introduced and became commercially available *in the midst of the composition of this thesis*, a development not anticipated at the onset of this work. Unfortunately, the principles of its inner functionality are concealed by the manufacturer.

## 2.7 Conclusion

In conclusion, the foundational aspects of driving automation have been thoroughly examined. Initially, the driving process itself was scrutinized, presenting a classification of automation levels based on the roles of human and machine. Furthermore, critical facets of functional safety, including automotive security integrity levels (ASILs) and fault tolerance time interval (FTTI), were analyzed, facilitating the application of the developed software in testing automotive component software.

Subsequently, the role of the general E/E architecture of a vehicle in the context of driving automation was investigated. This involved examining the segregation of the vehicle's internal

electronic control units (ECUs) into domain-specific functionalities, concluding that achieving higher levels of automation is contingent on establishing either a domain-centralized or a zonal architecture.

A comprehensive analysis of automotive networking, which facilitates communication between separate modules (domains), was then conducted. During this analysis, various technologies used in modern vehicles were examined and classified based on their characteristics. A key finding is that achieving advanced levels of driving automation today is feasible primarily through the use of Automotive Ethernet as the communication technology, owing to its bandwidth, particularly beneficial under the substantial network loads characteristic of highly automated vehicles.

The next logical phase entailed an in-depth examination of the structure and types of Automotive Ethernet, as well as the role of the ISO/OSI model in organizing communication through this technology. Each layer of the model was explored, along with the primary protocols utilized at each level.

In the context of exploring real-time data manipulation methods in Automotive Ethernet, the organization of communication of individual software components via the AUTOSAR framework was analyzed. This analysis included examining key abstract data units involved in data transmission (signals, signal groups, PDUs, PDU groups), their internal structure, properties, flow and some significant actions performed with them during data transmission (e.g., multiplexing). The primary conclusion of this analysis is that manipulating signals in the Automotive Ethernet network is achievable through direct modification of the necessary bits transmitted in the payload of internet packets. AUTOSAR XML files, describing communication between different software component interfaces and some details pertaining to communication security, can be used to identify these bits.

One of the final stages of analysis addressed potential attacks and existing protection mechanisms in in-vehicle networking from three perspectives: the ISO/OSI model, AUTOSAR, and most importantly, functional safety. The analysis led to the identification of mechanisms to circumvent featured protections, specifically: the necessity of real-time recalculations of FCS, CRC, MAC, and possibly BZ (to prevent PDU obsolescence), and their subsequent embedding in the packet payload at the appropriate location.

The final stage briefly reviewed existing solutions in the domain under consideration.

In conclusion, manipulating signals in the Automotive Ethernet network in a manner that remains undetected by the communicating parties (refer to section 1.2) is theoretically feasible under the specific conditions. The primary requirements for its successful execution include having the adjusted environment, accurately determining the location data of the manipulated signals (their transmission mode is not important), replacing the values of selected bits, detecting protection mechanisms based on ARXML files, and circumventing them in real-time within the minimal possible timeframe to comply with ASIL's FTTI. Therefore, the main potential bottlenecks (most important issues) to consider are the analysis of ARXML files and the direct interaction with Ethernet traffic.

# Requirements Synthesis

In this chapter, the crucial needs and specifications are systematically dissected in order to outline the scope for the further development of a software capable of signals manipulation within the Automotive Ethernet network. It is divided in four sections, each addresses a specific aspect.

First, section 3.1 (Functional Limitations) specifies inherent constraints and boundaries on the system’s capabilities. It identifies potential limitations in manipulating signals within the Automotive Ethernet framework derived from the Chapter 2.

Section 3.2 (User Requirements) details the expectations and preferences of end-users interacting with the system, taking the mentioned limitations into account. It includes considering of user experience, interface preferences, and any specific user-driven functionalities. Generally it establish a user-centric approach to inform design and development, by providing the essential set of the use cases.

Section 3.3 (Hardware Requirements) specifies the necessary hardware components and configurations for the system, considering speed, processing power, and compatibility with existing Automotive Ethernet infrastructure. It ensures a comprehensive understanding of the hardware prerequisites for optimal system performance.

Finally, section 3.4 (Software Requirements) defines the *functional* and *non-functional* requirements derived from user requirements & use cases. It defines the essential software attributes and functionalities needed for seamless operation.

To conclude, this chapter serves as a pivotal guide for shaping the system’s design and functionalities, aligning them with the practical and operational necessities inherent to the Automotive Ethernet landscape.

## 3.1 Functionality Limitations

To begin with, as it turns out, the creation of the software capable of a comprehensive signals manipulation goes far beyond the scope of this master’s thesis due to the large amount of circumstances that would demand a consideration. Moreover, since the establishment of Automotive Ethernet network within a car, and of AUTOSAR in particular, encompasses a variety of manufacturer specific nuances, the developed system cannot be universally applicable.

Therefore, the overall system developed shall be treated as a *functional prototype* able to perform the featured functionality. Hereinafter, the development of a system for signals manipulation on Automotive Ethernet shall be based on the following provisions:

**L1: The system is strictly confined to be software-based only.** No integration with specialized hardware or equipment facilitating signal manipulation is taken into consideration and *initial* compatibility is limited to an usual PC.

- L2: The system shall not be fully autonomous.** It order to operate, it shall partially rely on the information provided by the user.
- L3: The system’s applicability is limited to inter-ECU communication exclusively.** Manipulation of signals between different SWCs located in one ECU requires additional integration into AUTOSAR RTE, which is deemed to be unnecessary in terms of this work.
- L4: The Automotive Ethernet network employed shall not incorporate any specialized traffic protection protocols, such as IPsec, MACsec, TLS, gPTP or any alternative solutions, including those developed by the manufacturer.** Bypassing the protection provided by one or more of these mechanisms could be challenging and is a separate issue that goes beyond the scope of this thesis.
- L5: The system shall be compatible merely with IEEE 802.3 Ethernet standard.** Compliance with other standards (namely IEEE 1722 in context of AVB/TSN) is not a common case and requires additional efforts, theoretical background, etc.
- L6: The system shall support only partial AUTOSAR SecOC profile 2 circumvention.** Since the Secret Key is being managed independently by a compound of AUTOSAR framework with different scenarios possible (timing for the secret key renovation), the system shall assume renovation timing dependent on the power cycle of a car (no possible Secret Key update on the sender/receiver side at the time of a system operation). Bypassing of AUTOSAR SecOC profile 1 shall not be supported because at the time of a system operation it is impossible to obtain the actual last received FV on the receiver side if earlier communication between sender and receiver had been already held before the system started running. Additionally, the *simplified* mapping of secured I-PDU to L-PDU presumed for the sake of simplicity, assuming that the actual MAC is stored inside the L-PDU payload as signal.
- L7: The system shall not support the usage of Update-bits by AUTOSAR COM.** Update-bits make the recipient to be notified of changes in signal values in advance, which would cease the ability to interfere into the communication. This is because even after bypassing all the security mechanisms, the tagret SWC would treat the received data as corrupt at the application level due to the absence of prior notification on pending signal changes. The circumvention of this optional mechanics would demand additional measures that are out of scope of the thesis’ problematics.

## 3.2 User Requirements

From the user’s perspective, the developed system shall be applicable in *system testing* (refer to section 5.1) of the entire vehicle with domain centralized or zonal E/E architecture. The developed system is needed to assist in application of test methods (required by ISO 26262, see section 2.1.2) to individual SWCs (ECUs) interacting with others, for the purpose of evaluation of their performance according to the specified use cases.

Despite the assessment of the performance of individual SWCs/ECUs is not covered by the scope of the very thesis, the system shall rather provide the means for the very assessment. Since the assessment can be automated or performed manually, system shall support both command line and GUI interaction with the user. The system should be capable of manipulation with data sent by a sender ECU before the reception by a receiver ECU, without any impact on the communication of other parties if necessary. When no manipulation with data is performed, the system shall act as a transparent gateway. Both functionalities should be carried out with a minimal possible delay.

In order to achieve it, the system shall be capable of modification of individual signals (whether they are the part of a signal group or not) within a PDU transmitted over the network

in such a way, that on arrival to the receiver it is not treated as a corrupted one on application level or by AUTOSAR SecOC module (if implemented). This, in turn, namely includes setting a signal value to the provided one (test use case specific), modification of CRC and FV of PDU (optionally), circumvention of the supported AUTOSAR SecOC profile (optionally), recalculation of Ethernet FCS & other associated checksums and resending the Ethernet packets in real-time, satisfying the signal delivery delay tolerance requirements (ASIL related FTTI).

Since the communication in such case is being performed via physical media, the `.arxml` file providing the essential information on the layout of featured signals within PDUs (whether multiplexed or not) in the Ethernet packet is to be used; there is no need in information about the signals & PDUs layout at upper abstraction layers of AUTOSAR environment. If modification of application level CRC and FV or AUTOSAR SecOC bypassing is necessary, then the details lacking in proper `.arxml` file shall be provided by the user manually, particularly polynomial used for in-PDU CRC calculation and Secret Key for AUTOSAR SecOC bypassing. Moreover, user is responsible for defining the exact actions the system has to perform (to filter packets or to modify signals) depending on the external context of system testing mentioned earlier.

In addition, the system shall be capable of memorizing used configurations in order to ease the overall testing process it is to be used in.

### 3.2.1 Use Cases

Overall, the behavior of the developed software shall match the following use cases:

**UC1: Changing the values of individual signals.** System acts as a transparent gateway upon start-up, performing no traffic filtering or modification. User specifies the signal name, `.arxml` file containing its specifications, the new value to be set and the duration of actioning (whether in cycles or time units). If user is aware of security mechanisms applied to the specified signal, the additional data is provided (Polynomial for CRC calculation and/or Secret Key for AUTOSAR SecOC 2 circumvention). The proper configuration is created<sup>1</sup>. User activates the desired configuration, which is then stored. The program starts its execution according to the user command, begins manipulation with signals and bypasses all the security mechanisms based on the user input. If the duration of the configuration has expired (in cycles or milliseconds), it is no longer applied. The program stops its operation according to the user command, becoming a transparent gateway again.

**UC2: Filtering the inter-ECU communication by omitting the specified elements.** System acts as a transparent gateway upon start-up, performing no traffic filtering or modification. User has the capability to define filtering criteria for communication, thereby transforming the program into an opaque gateway (proxy). Source & destination IP addresses as well as ports, and PDU id can be defined to outline the filtering criteria. The proper configuration is created; user activates the desired configuration, which is then stored. Program starts its execution according to the user command and filters the traffic excluding the specified packets from the communication bus<sup>2</sup>. If the duration of the rule has expired (in cycles or milliseconds), it is no longer applied. The program stops its operation according to the user command, becoming the transparent gateway again.

**UC3: Activating, deactivating, importing, deleting and exporting of utilized configurations.** Program stores all the created configurations (both at a run-time and between usage sessions). User has an opportunity to upload several configurations to switch between

---

<sup>1</sup> *Configuration* or *rule* represent a collection of information required to modify the signal according to the request of a user.

<sup>2</sup> This feature ensures that only traffic meeting the specified criteria is permitted to traverse through the network. It is especially expedientary when testing the ECU in isolation during the operation of a vehicle (e.g. physical cable damage), as well as when testing ASIL related FTTI.

them later (not to enter new signal value and not to process `.arxml` file, if some configurations have to be repeated). User has an opportunity to delete a specific configuration, get all previously created configurations, update, activate or deactivate the selected ones.

While the program has limited applications due to its specificity, the described scenarios of its usage and behavior fully correspond to situations in which it would be applied during the real testing sessions<sup>3</sup>. They serve as the basis for specification of the desired functionality of the system in details.

### 3.3 Hardware Requirements

In accordance with functionality limitation L1, the developed system shall not require any specific HW supplements. Therefore, HW requirements for the system are merely bounded by the presence of a PC with Linux operational system installed and two Ethernet Network Interface Cards (NICs) available.

The exact impact of quality characteristic of its components (processor clock speed, number of physical and logical cores, RAM, etc.) shall be evaluated in section 5.1.

### 3.4 Software Requirements

The software requirements for the system for manipulation with signals on Automotive Ethernet (later – 'system') are derived from user (or customer) requirements, use cases and theoretical background presented in the previous chapter. They represent the desired functionality of the software to develop and can be divided into *functional* and *non-functional*.

The very requirements are specified in the following subsections and must be interpreted in the context of the previously established functionality limitations and *not in isolation from them*. It must also be noted, that the exact implementation of some requirements depends on the environment the system is intended to operate in due to manufacturer-specific nuances.

#### 3.4.1 Functional Requirements

**FR1:** System has to be capable of changing the value of specified signal(s).

**FR2:** System has to be capable of automatic detection and circumvention of in-PDU Alive Mechanism.

**FR3:** System has to be capable of automatic detection and computation of in-PDU CRC based on provided polynomial.

**FR4:** System has to be capable of automatic recomputation of in-packet checksums.

**FR5:** System has to be capable of automatic detection and circumvention of AUTOSAR SecOC (profile 2) based on provided Secret Key.

**FR6:** System has to be capable of automatic exclusion of specified packets (depending on source & destination IPv6 address, port and PDU id) from the communication bus.

**FR7:** System has to be capable of obtaining all necessary signal, PDU and Ethernet frame specifications from the `.arxml` file provided.

---

<sup>3</sup>The provided user requirements and use cases were compiled with the support of the Vehicle Testing Department of Porsche Engineering (Porsche Engineering Services s.r.o.), located at Radlická 714/113a, 158 00 Prague 5, Czech Republic.

**FR8:** System has to be capable of run-time activation, deactivation, deletion, importing, exporting, updating and long-term maintenance of used configurations.

**FR9:** System has to provide a user with both GUI and command line operation possibilities.

► Note 3.1. The inclusion of the functionality limitation L6 and the functional requirement FR5 is predicated on the examination of the potential to circumvent *certain* protection mechanisms associated with AUTOSAR. Should this be successfully executed under the *simplified conditions* established in L6, it would also indicate the possibility of bypassing this protection mechanism in its full-scale model.

### 3.4.2 Non-functional Requirements

**NFR1:** The overall system performance has to comply with specified fault tolerant time interval, depending on the use case.

## 3.5 Conclusion

This chapter establishes a foundational framework for developing a software system for signal manipulation within Automotive Ethernet networks, focusing on practicality and operability in line with Automotive Ethernet and AUTOSAR standards. The system, conceptualized as a functional prototype, adheres to clearly defined and logically derived functionality limitations, ensuring realistic expectations within the constraints of these frameworks and thesis in general.

User requirements were compiled with the support of Porsche Engineering and are tailored to facilitate system testing, emphasizing features like signal manipulation and traffic filtering, crucial for evaluating SWC/ECU performance and ISO 26262 compliance.

Minimal hardware requirements ensure broad accessibility and ease of deployment, while the software requirements are comprehensively outlined, encompassing both functional capabilities and important non-functional aspects like system performance in context of fault tolerance. The requirements for the final software are intentionally left somewhat vague, providing the author with the freedom to implement while focusing solely on the key functionalities. The system's adaptability and user-centric design are key to its usability in diverse automotive testing scenarios.

Overall, this thesis outlines a software system that is both specialized and adaptable, poised to be a significant tool in Automotive Ethernet testing and analysis. The system's relevance and potential impact in the automotive industry are underscored by its alignment with current and evolving industry standards.





# Software Design & Implementation

This chapter delves into the critical phases of software design and implementation, encapsulating the intricacies of realization of stated functional requirements. First, the general high-level architecture of a system is being designed according to previously derived & analyzed software requirements (refer to section 4.1), all the design & technology solutions are justified and explained (see section 4.2). Second, the detailed design of selected individual components is being presented and discussed in section 4.3. Consequently, several important insights into the actual implementation phase are provided, unveiling the real-world challenges faced.

## 4.1 Architectural Design

Upon examining the functional requirements compiled for the software in section 3.4.1, it becomes evident that the desired end functionality, particularly in the context of internet packet processing, can be categorized into three fundamental groups: '*Preparation*', '*Action*', and '*Miscellaneous*'. The '*Preparation*' category encompasses a set of measures vital for the successful execution of tasks within the '*Action*' category, which represents the software's ultimate objective. '*Miscellaneous*', on the other hand, comprises functional capabilities essential for facilitating user convenience and controllability of a system.

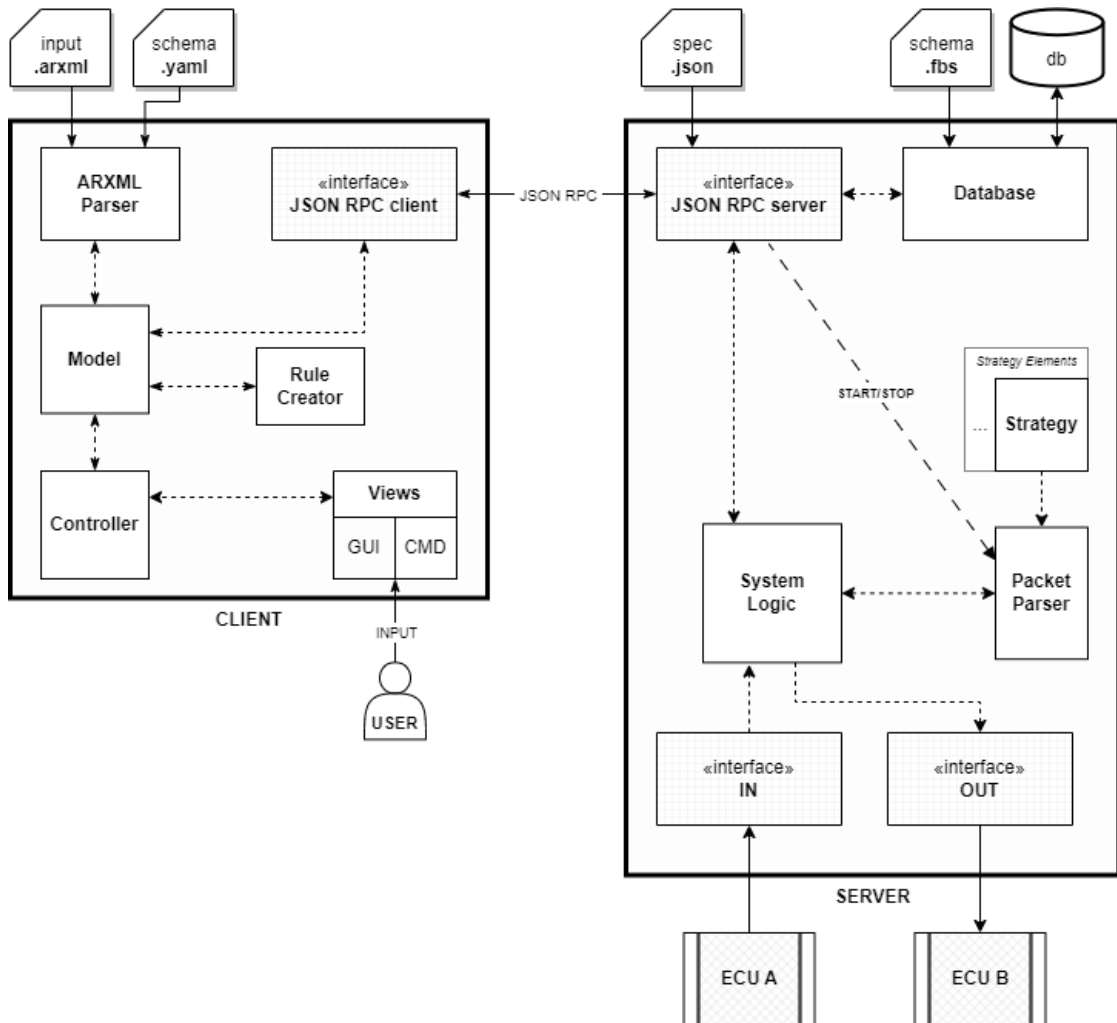
This is particularly apparent when examining requirements FR2, FR3, and FR5, where '*automatic detection*' falls under '*Preparation*', while '*calculation*', '*circumvention*' etc. belong to '*Action*'. The very signal specification can be considered as '*Preparation*' as well. Furthermore, requirement FR7 is exclusively related to '*Preparation*' as it does not reflect the software's end goal, but rather the prerequisite for achievement of it. Requirements FR1 (partially), FR4, and FR6 directly correspond to the '*Action*' category. The remaining functional requirements of the software align with the '*Miscellaneous*' category.

This logical division of software requirements implicitly suggests a modular approach to its design and implementation. In turn, this approach enables the application of many principles characteristic of successful software design, such as '*separation of state*', '*separation of concerns*', '*data version transparency*' and '*action version transparency*' [64] among others. Concurrently, the further proposed design strives to achieve high cohesion within modules and low coupling between them in order to further enhance the possible software's maintainability and, perhaps, scalability, aiming to optimize these aspects, however, without making them the paramount concern. The employment of denoted principles to a software system will benefit it by reducing the combinatorial effects, therefore steering it closer to a *normalized* one.

The overall design choice favors a global *client-server* architecture; the developed software is suggested to be named 'Automotive Ethernet Protocol Injection Logger', or AEPRIL. Although the selected architecture is less optimal in terms of achieving low coupling and high cohesion compared to a microservices architecture, nevertheless it offers greater flexibility than a monolithic approach.

Furthermore, such a design facilitates the independent development, maintenance, enhancement, and testing of client and server components, provided that a predefined client-server interface is established. This allows the final system to be divided into two parts: one responsible for direct interaction with the user, receiving information, and processing data prior to the execution of requested actions (Preparation – client), while the other carries out these actions (Action – server). At the same time, the implementation of Miscellaneous requirements is shared among them.

Such a division naturally delineates the system's inherent bottlenecks: the processing of .arxml files and the direct intervention in Automotive Ethernet traffic (refer to section 2.7). In this way, the real-time Automotive Ethernet packet processing (which must be as swift as possible) is undertaken by the server-side of a system and is decoupled from the parsing of ARXML file, which could take relatively significant amount of time.



■ **Figure 4.1** Architecture of a system for signals manipulation on Automotive Ethernet.

After AUTOSAR XML file is processed by client, all the information required for the proper handling of traffic is suggested to be sent to client via JSON RPC v2.0 protocol. Moreover, it in turn implies the possibility of independent installation and usage of each component on discrete interconnected devices, enabling the remote control of system operation.

The general architecture of a system is represented at Figure 4.1. The client and server architecture comprises several intuitively understandable abstract components, each with specific responsibilities, reflecting the 'separation of concerns and state' principles for independent management of software aspects. The data flow and internal action sequence are explicitly outlined in the system's schematic, illustrating data movement and conforming to 'data and action version transparency' principles, thereby clarifying the process's evolution over time.

The structure of both client and server, as well as the very client-server interface and external dependencies are described in details in sections 4.3.2, 4.3.3 and 4.3.1 respectively.

## 4.2 Technologies & Libraries

Implementation of any software necessitates a predefined development means & methodology and a clear understanding of the environment where the software will be deployed. This includes identifying key programming languages, external libraries used, the operating environment (platform), and the overall paradigm for project implementation.

In this case, the server is developed exclusively for the Linux platform, aligning with the objectives set for this work. This choice facilitates the use of the Linux Kernel for near-direct interaction with network card drivers. Given the requirement for rapid processing of large volumes of raw data (streams of internet packet bytes) and the limited range of functionalities to be implemented, C 17 (ISO/IEC 9899:2018) and C++ 20 (ISO/IEC 14882) are identified as the most suitable programming languages.

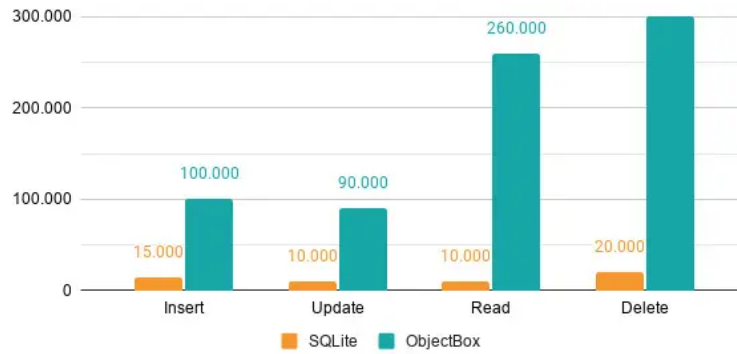
Conversely, the client, responsible for direct interaction with user, requires an appropriate graphical interface and the capability for an alternative command-line control. Additionally, the management of large data sets with inherent abstraction (such as AUTOSAR XML files) demands a higher-level approach. Python 3.11 is chosen for the client to ensure cross-platform installation and usage, along with access to a wide range of libraries that facilitate development.

In addition, the development process employs both object-oriented and, to some extent, functional programming paradigms. The client development was undertaken using JetBrains PyCharm Integrated Development Environment (IDE) 2021.1, while Microsoft Visual Studio Code 1.55 and CMake build system framework 3.20 were employed for server development. The main libraries used, the reasons for their selection (apart from personal experience), and the ways of their application are detailed in the following subsections.

### 4.2.1 ObjectBox

*ObjectBox* is a high-performance NoSQL database, which excels in rapid data processing and operational efficiency (refer to Figure 4.2), extremely crucial for high-performance computing, embedded systems, etc. It supports a range of programming languages (C/C++, Java, Python, Swift, etc.) making it versatile across various software applications. Binding code for ObjectBox APIs is generated by ObjectBox Generator according to predefined `.fbs` schema, describing the objects that are to be "stored". Distinguished by its low latency, high throughput, and compact size, ObjectBox is particularly well-suited for resource-limited devices.

The operational framework of ObjectBox uses an object-oriented data storage model, aligning with object-oriented programming to improve data access and manipulation. Moreover, ObjectBox maintains ACID-compliant transactions, ensuring data reliability, particularly in unpredictable system scenarios. In addition, it allows to build custom queries for tailored data



■ **Figure 4.2** ObjectBox CRUD Operations per second in comparison with SQLite; adapted from <https://objectbox.io/>.

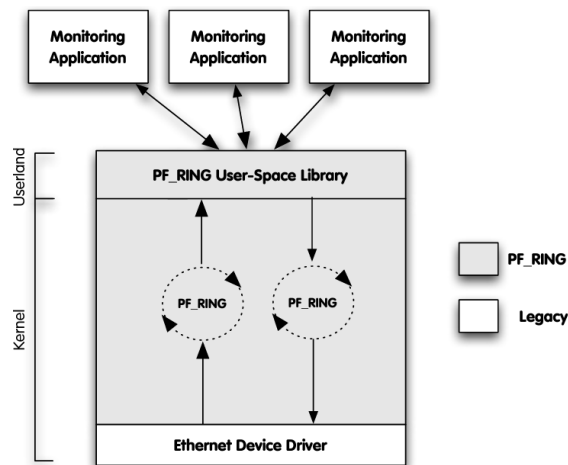
retrieval needs, as advanced indexing significantly cuts down search and retrieval times, enhancing query performance.

The ObjectBox is notably open-sourced under the Apache 2.0 license, which makes it freely available for both commercial and personal use.

These capabilities render ObjectBox particularly suitable as a database solution for the server side of the end system.

## 4.2.2 PF\_RING

*PF\_RING* is an advanced high-performance packet processing framework, designed to provide means for network monitoring, packet capture, filtering and analysis. Despite several usage-specific *PF\_RING* versions exist (the *Vanilla PF\_RING* is to be used), the framework core comprises an accelerated kernel module and a user-space SDK supplied with the proper API. The general architecture of the framework is illustrated on Figure 4.3.



■ **Figure 4.3** PF\_RING's architecture.

The framework itself supports a wide range of functionality, including the ability to bind applications to various network interfaces, such as physical and RX queues. Additionally, an efficient packet processing mechanism, where packets are transferred to a pre-allocated memory ring (circular buffer), minimizes overhead, which makes it crucial for high-speed networks. The

framework's support for an array of packet filtering options, as well as packet parsing functionality, capable of extracting metadata from network layers (ISO/OSI layers 2-4), enables precise traffic analysis. Notably, PF\_RING's packet reflection feature and hardware-based filter compatibility are essential for real-time packet processing, while its clustering mechanism allows for distributed packet processing, ensuring efficient traffic management in high-volume network scenarios.

PF\_RING distributes its kernel module and drivers under the GNU GPLv2 license, while its user-space PF\_RING library is under LGPLv2.1, all available in source code format, making it available for both personal and commercial use.

In summary, PF\_RING's comprehensive capabilities make it a potent tool for Automotive Ethernet packet capturing and their processing using CPU, addressing the needs for speed and efficiency, which make PF\_RING a versatile deployment for the system's server side.

### 4.2.3 PcapPlusPlus

*PcapPlusPlus* library is a comprehensive tool designed for network packet capturing, parsing, crafting, and analysis. It provides a unified C++ API to interact with various packet capture engines, simplifying their complexity and offering a common platform for a wide range of network-related operations. Among all others, it supports several key features especially relevant in terms of this work.

First, the library supports packet parsing and crafting, by providing advanced capabilities in packet filtering, as well as in both analyzing the details of network packets and creating or modifying them. PcapPlusPlus supports reassembling fragmented network packets at both the IPv4/IPv6 (Network layer) and TCP (Transport layer) levels, facilitating the handling of large data chunks across network protocols.

Second, PcapPlusPlus excels in packet capture and sniffing (intercepting and logging network traffic) by supporting multiple packet capture/processing engines such as libpcap, WinPcap/Npcap, Intel DPDK, and notably PF\_RING (refer to section 4.2.2). The library supplies a user with an ability to utilize previously mentioned Vanilla PF\_RING engine via provided wrapper, and therefore eliminates the necessity to use it directly. In justification of it, for instance, an Ethernet packet captured by PF\_RING can then be uniformly handled by the PcapPlusPlus API, making it possible to process it via functionality offered by another packet processing engine.

PcapPlusPlus is released under the Unlicense license, making it freely available for both personal and commercial use.

In summary, PcapPlusPlus stands out as a versatile and efficient solution for processing the ethernet traffic at the server component of the system.

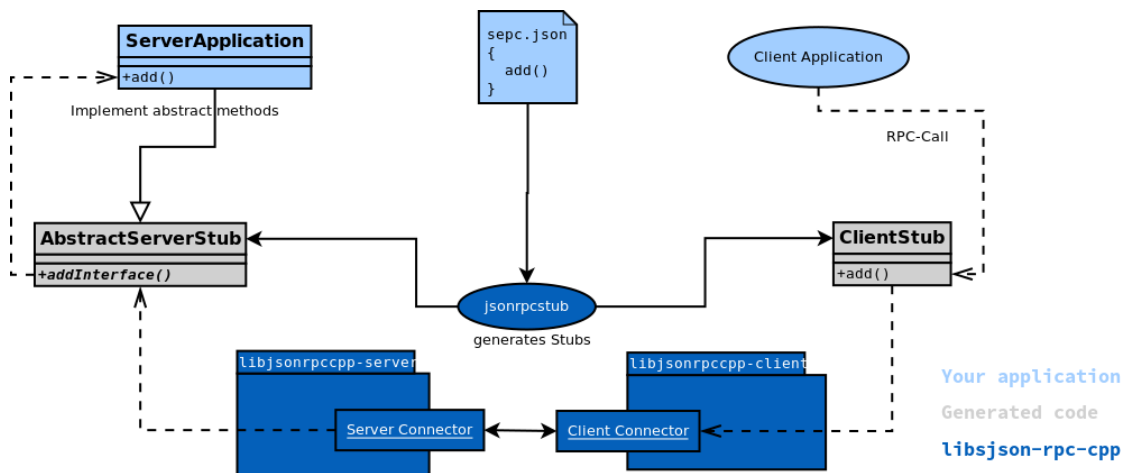
### 4.2.4 Libjson-rpc-cpp

*Libjson-rpc-cpp* framework offers essential support for implementing JSON-RPC (Remote Procedure Call) in C++ applications, aligning with the standards of JSON-RPC 2.0 and partially with JSON-RPC 1.0. Libjson-rpc-cpp automates the generation of stub classes for both client and server sides of RPC applications (refer to Figure 4.4) according to the provided JSON specification.

The framework includes pre-built HTTP and TCP servers and clients. This feature provides straightforward and efficient interfaces for JSON-RPC applications, significantly reducing development time. It extends its utility by offering build support for Linux, which is especially relevant in context of the server compound.

The adoption of the MIT License promotes extensive use (both personal and commercial) and modification of the framework.

Summarizing, libjson-rpc-cpp is a lightweight and universal solution for JSON-RPC based communication that will be utilized solely on the server side of the resulting system.



■ **Figure 4.4** Generation of libjson-rpc-cpp stub classes and their usage.

## 4.2.5 PyQt6

To begin with, Qt is set of cross-platform C++ libraries that implement high-level APIs for accessing many aspects of modern desktop and mobile systems. These include location and positioning services, multimedia, NFC and Bluetooth connectivity, a Chromium based web browser, as well as traditional UI development, etc.

PyQt6 is a comprehensive set of Python bindings for Qt v6. It is implemented as more than 35 extension modules and enables Python to be used as an alternative application development language to C++ on all supported platforms. PyQt6 may also be embedded in C++ based applications to allow users of those applications to configure or enhance the functionality of those applications.

PyQt6 is released under the GPL v3 license and under a commercial license that allows for the development of proprietary applications.

In context of this thesis, PyQt6 is to be used in GUI development and multithreading support for client side system compound.

## 4.2.6 lxml

The *lxml* XML toolkit integrates Pythonic bindings for the libxml2 and libxslt C libraries, enhancing the ElementTree API with a focus on Python's simplicity and C's performance. It constructs on the libxml2 tree, offering improved functionality but at the cost of higher maintenance due to the dynamic generation of Python node representations.

Moreover, the primary limitation of lxml is exactly its reliance on the complex tree model of libxml2; this model complicates tree construction and restructuring processes. However, it also provides advanced features like parent pointers and, especially, XPath support, which enables querying of XML documents.

In addition, serialization is a key strength of lxml, operating at the C level, which results in significantly faster performance compared to other ElementTree versions in Python. The toolkit is particularly efficient with UTF-8 encoding. Exact benchmarks are available on the official webpage of lxml.

The lxml library is shipped under a BSD license, while libxml2 and libxslt2 itself are shipped under the MIT license, which allows both commercial and personal use.

Overall, lxml is an acceptable solution for XML querying & parsing on the client side of the end system.

## 4.3 Detailed Design & Implementation

This section focuses on the detailed design and implementation of a system for signals manipulation on Automotive Ethernet. First, the client-server interface is established and described. Later, the architecture of the system's major compounds (client & server) is described separately, highlighting their key inner components and principles of their interaction. The design choices made during the development process are mainly explained, emphasizing their alignment with established best practices; information on challenges faced and how they were overcome is presented as well. Additionally, this section outlines the practical steps taken during the implementation phase, providing the featured insights into the source code of the developed software and the methodologies adopted.

This part of the thesis aims to provide a clear link between theoretical concepts and their application in a real-world scenario. Particularly, this section may be considered as a technical guide to the system developed.

### 4.3.1 Client-Server Interface

To establish client-server communication following the JSON RPC v2.0 standard, the *methods* (procedures) for remote invocation must first be defined. Since their definition is not dictated by any specific requirements, their precise delineation was conducted based solely on personal discretion and the envisioned functionality of the end system. The resulting interface is intended to provide means for basic control over the server-side of a system. It supports the remote procedure call of the following methods:

1. **TEST** — the method is employed to verify the connection between the client and the server; it performs no functionality. Essentially, its successful execution signals to the client the server's availability and current operational mode (refer to Table 4.1 for more details).
2. **START** — the method is used to activate the program's operational mechanism in accordance with the currently active rules. Table 4.2 provides the specification of this method.
3. **STOP** — the method is utilized for deactivating the operational mechanism of the program, consequently reverting it to a state of transparent gateway functionality. Detailed specification of this method is available in Table 4.3.
4. **UPSERT** — the method is used to either update existing data or insert new data if it does not already exist. The exact operation to be performed is determined by a server depending on the data received from client. The parameters of the method, along with their descriptions, as well as the results returned upon method invocation, are detailed in Table 4.4.
  - Note 4.1. The input parameters defined for the UPSERT method are deemed to be sufficient and, importantly, shall be construed as the precise specification of a rule itself.
5. **SET** — the method is designed to alter the status of a chosen rule, setting it to either "active" or "inactive". Detailed information about this method can be found in Table 4.6.
6. **GET** — the method is utilized for transmitting rules stored on the server to the client. It facilitates the transfer of either a single selected rule or all existing rules on the server in form of an array (depending on the parameters provided). A detailed specification of the method is presented in Table 4.5.
7. **DELETE** — the method is employed for the permanent removal of a selected rule (or all rules) from the server. Detailed specifications of this method are available in Table 4.7.

Name	Result	Params		
		Name	Type	Description
TEST	{ res ( <i>bool</i> ), message ( <i>string</i> ) }	—	—	This method has no input parameters.

■ **Table 4.1** JSON RPC TEST method specification.

It is important to mention that the specifications of the methods' results, as detailed in the referenced tables, are relevant only in scenarios where the procedure is invoked successfully. In the event of a failure, the standard JSON RPC v2.0 *error object*<sup>1</sup> is employed in the response of a method. A failure in this context may arise from a logical error, such as the impossibility of deleting a non-existent rule, or from a usage error, like passing an incorrect number of parameters.

Name	Result	Params		
		Name	Type	Description
START	{ res ( <i>bool</i> ), message ( <i>string</i> ) }	—	—	This method has no input parameters.

■ **Table 4.2** JSON RPC START method specification.

Name	Result	Params		
		Name	Type	Description
STOP	{ res ( <i>bool</i> ), message ( <i>string</i> ) }	—	—	This method has no input parameters.

■ **Table 4.3** JSON RPC STOP method specification.

To illustrate the practical application of JSON RPC, the following example demonstrates the invocation of the DELETE method using a curl command:

```
$ curl -X POST http://192.168.1.12:8080 -H "Content-Type: application/json" -d \
'{
  "jsonrpc": "2.0",
  "method": "DELETE",
  "params": {
    "id": 1
  },
  "id": 3
}'
```

<sup>1</sup>As defined in the protocol, the *error object* comprises the fields *code* (of type *int32*) and *message* (of type *string*), representing the error code and its corresponding explanation, respectively.



This curl command sends a POST request to the JSON RPC server, here hypothetically located at '192.168.1.100' on port '8080'. The request specifies the *DELETE* method, with parameters set to delete the rule with ID 1 from server. The 'id' field in the JSON body represents a client-defined identifier for this request, arbitrarily assigned the value 3.

Name	Result	Params		
		Name	Type	Description
UPSERT	<pre>{   id (int64),   message (string) }</pre>	mode	<i>bool</i>	Operational mode: <b>false</b> for packet filtering, <b>true</b> for signal modification.
		id	<i>int64</i>	Rule ID assigned by a server, or 0 by default.
		status	<i>bool</i>	Rule activity status: <b>true</b> if rule is currently active, <b>false</b> otherwise.
		duration	<i>int32</i>	Numerical expression of the duration of rule usage.
		duration_type	<i>string</i>	Type of the duration: milliseconds ("ms"), seconds("s"), minutes ("m"), cycles ("cyc") or infinite ("inf").
		src_ip	<i>string</i>	Source address of an IPv6 packet. If mode is <b>false</b> , empty string stands for any.
		src_port	<i>int16</i>	Source port of an IPv6 packet. If mode is <b>false</b> , -1 stands for any.
		dest_ip	<i>string</i>	Destination address of an IPv6 packet. If mode is <b>false</b> , empty string stands for any.
		dest_port	<i>int16</i>	Destination port of an IPv6 packet.
		pdu_id	<i>int16</i>	PDU ID: either ID of a PDU containing the desired signal, or a ID of a PDU within IPv6 packet that is to be excluded from traffic.
		signal_name	<i>string</i>	Name of a signal.
		signal_start_bit	<i>int16</i>	Absolute offset in bits from the beginning of the SDU (PDU) to the beginning of a signal.
		signal_length	<i>int16</i>	Length of the signal in bits.
		new_value	<i>uint64</i>	New value to of a signal to be set.
		polynomial	<i>uint8[]</i>	Polynomial required for in-PDU CRC calculation provided by a user.
		secret_key	<i>uint8[]</i>	Secret key required for AUTOSAR SecOC 2 circumvention provided by a user.
		bz_start_bit	<i>int16</i>	Absolute offset in bits from the beginning of the SDU (PDU) to the beginning of the in-PDU Alive counter.
		bz_length	<i>int8</i>	Length of the in-PDU Alive counter in bits.
		crc_start_bit	<i>int16</i>	Absolute offset in bits from the beginning of the SDU (PDU) to the beginning of the in-PDU CRC.
		crc_length	<i>int8</i>	Length of the in-PDU CRC in bits.
mac_start_bit	<i>int16</i>	Absolute offset in bits from the beginning of the SDU (PDU) to the beginning of the in-PDU MAC field.		
mac_length	<i>int8</i>	Length of the in-PDU MAC field.		
protocol	<i>bool</i>	Protocol if the IPv6 packet: <b>true</b> for UDP, <b>false</b> for TCP.		

■ **Table 4.4** JSON RPC UPSERT method specification.

► Note 4.2. Throughout this thesis, the *BZ* abbreviation (or *BZ counter* respectively), derived from the German term *'Botschaftszähler'*, will be used to denote the in-PDU alive counter. This terminology is adopted to maintain consistency with industry standards and ease of reference.

Name	Result	Params		
		Name	Type	Description
GET	[ { mode ( <i>bool</i> ), id ( <i>int64</i> ), status ( <i>bool</i> ), duration ( <i>int32</i> ), duration_type ( <i>string</i> ), src_ip ( <i>string</i> ), src_port ( <i>int16</i> ), dest_ip ( <i>string</i> ), dest_port ( <i>int16</i> ), pdu_id ( <i>int16</i> ), signal_name ( <i>string</i> ), signal_start_bit ( <i>int16</i> ), signal_length ( <i>int16</i> ), new_value ( <i>uint64</i> ), polynomial ( <i>uint8[]</i> ), secret_key ( <i>uint8[]</i> ), bz_start_bit ( <i>int16</i> ), bz_length ( <i>int8</i> ), crc_start_bit ( <i>int16</i> ), crc_length ( <i>int8</i> ), mac_start_bit ( <i>int16</i> ), mac_length ( <i>int8</i> ), protocol ( <i>bool</i> ) } ]	id	<i>int64</i>	ID of a rule to obtain from server. If ID is set to -1, all the rules stored are passed to client.

■ Table 4.5 JSON RPC GET method specification.

Name	Result	Params		
		Name	Type	Description
SET	{ res ( <i>bool</i> ), message ( <i>string</i> ) }	id	<i>int64</i>	The ID of the selected rule, the status of which needs to be changed.
		status	<i>bool</i>	The desired status to which the current status of the rule should be changed. If set to <b>true</b> , the rule becomes active; if <b>false</b> , it becomes inactive.

■ Table 4.6 JSON RPC SET method specification.

Name	Result	Params		
		Name	Type	Description
DELETE	{ res ( <i>bool</i> ), message ( <i>string</i> ) }	id	<i>int64</i>	The ID of the rule that is to be deleted. If ID is set to -1, all the rules stored on the server are to be deleted.

■ **Table 4.7** JSON RPC DELETE method specification.

The client-server interface showcased here serves as an initial prototype meeting the intended functional requirements of the developed system (FR8 in particular) and having a potential for future enhancements. It is designed without session establishment capabilities, allowing simultaneous access to server by multiple clients. While this design enables widespread use, it inherently makes the system more susceptible to DoS/DDoS attacks. However, addressing these security concerns, along with the implementation of explicit communication protection, falls beyond the scope of this thesis.

### 4.3.2 Client

In the initial stages, the development of the client-side of the system entails the implementation of requirements falling under the "Preparation" category as previously acquired. This encompasses the collection of information from the user, its appropriate processing, and augmentation to ensure compliance with the pre-defined client-server interface and its subsequent utilization. Each of these responsibilities is allocated to one of the client's abstract components, signifying that each component may consist of zero or multiple Python modules.

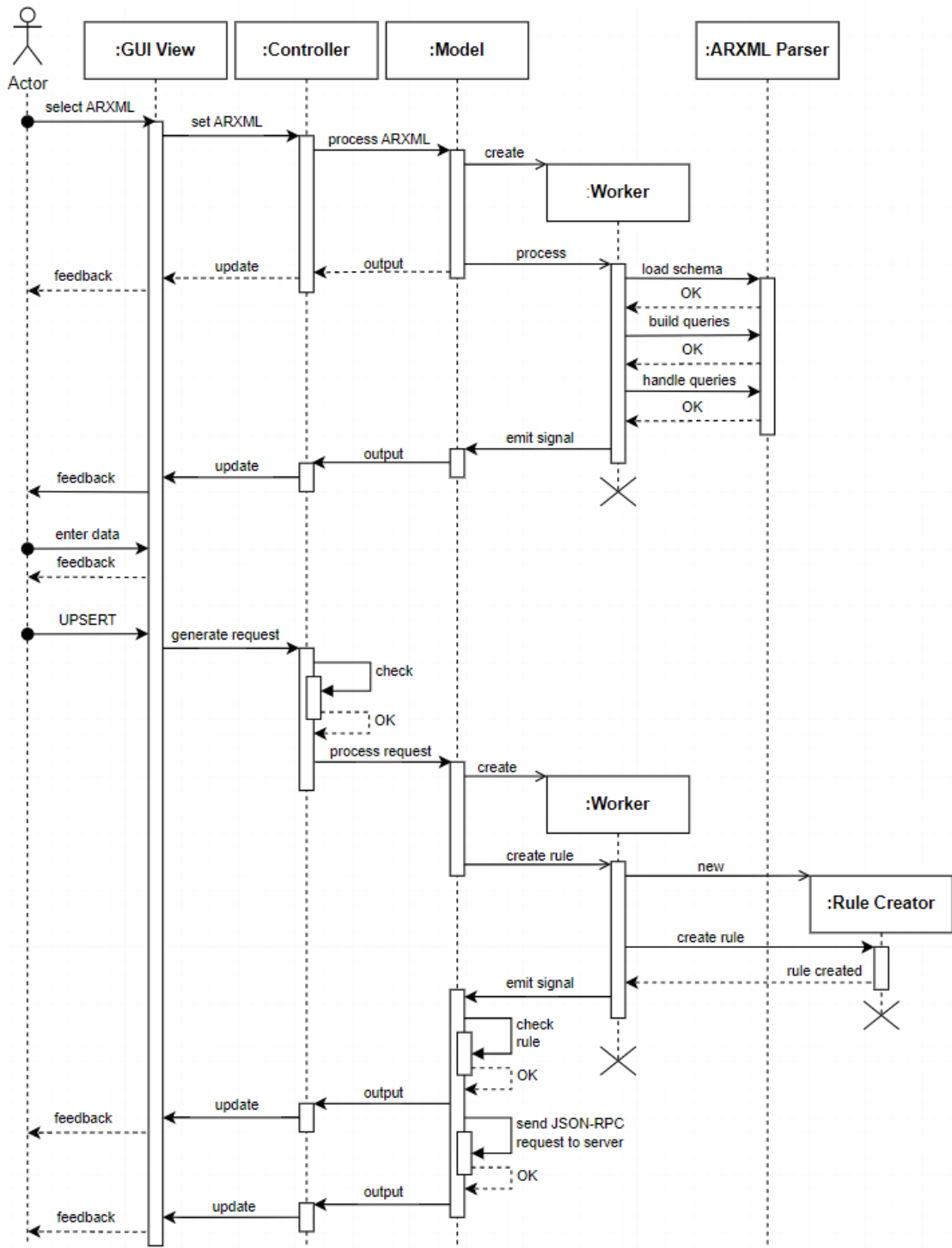
The client-side system architecture is distinctly structured, integrating a viable implementation of the Model-View-Controller (MVC) pattern with its abstract constituents – Model, View(s), and Controller. The adoption of the MVC pattern is strategically advantageous; it abstracts the program logic from the user interface, a design choice aligned with and justified by the functional requirement FR 9. In addition to the MVC framework, the architecture includes abstractions for an ARXML Parser, Rule Creator, and a JSON-RPC Client.

The design of the client-side part of the system is seamlessly connected to multithreaded programming. It involves a dynamic assignation of threads based on the type of selected View. The View's nature, whether it is a console interface or a graphical one, directly affects how many threads the system is to utilize. The program determines which View to use and how many threads to allocate depending on the command-line arguments given at startup.

► **Note 4.3.** The detailed information about the command-line arguments is available with using the `--help` argument at the time of program startup. Importantly, the supported command-line arguments mainly comply with rule attributes, outlined in section 4.3.1, and serve for their setting.

In cases where the system operates utilizing a GUI View, it is imperative to allocate an additional thread. This allocation is pivotal in maintaining the responsiveness of the interface. Conversely, in the configuration where the Console View is employed, the system's architecture allows the use of a singular thread for the sequential execution of individual tasks.

The primary operational process of the system's server component is depicted in Figure 4.5, while the essential constituents of the system are illustrated in Figure 4.1. The operational process of successful JSON-RPC request creation comprises several key stages, such as *arxml processing*, *input retrieval*, *rule creation* and *request submission* which are handled in one-by-one manner. A detailed exposition of the intended functionalities of each acting component, along with their internal mechanisms, is provided in the respective subsections.



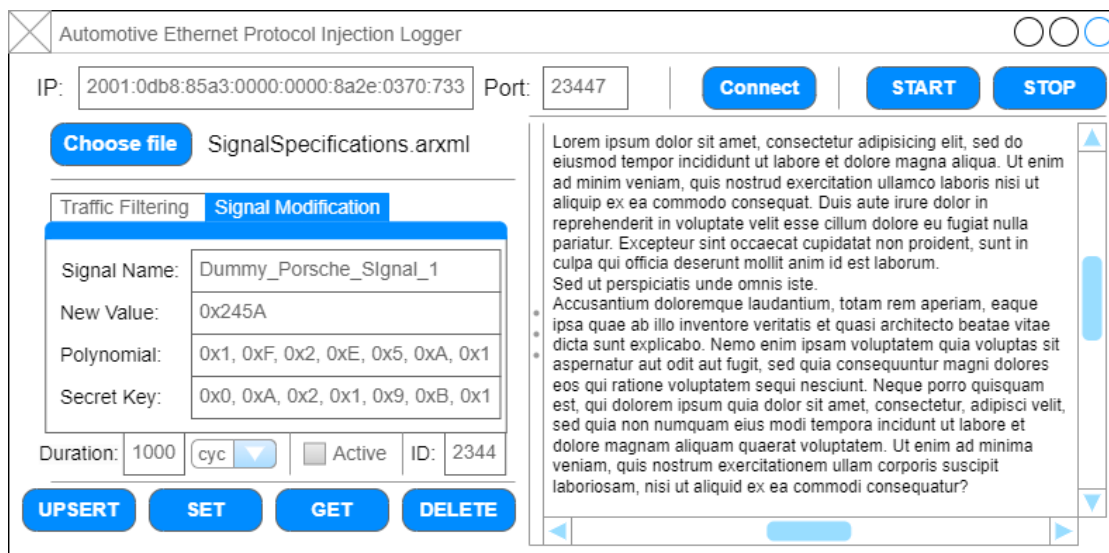
■ **Figure 4.5** Simplified UML sequence diagram, showcasing a successful internal process for ARXML parsing, acquiring of user data, rule creation and sending of UPSERT request to JSON-RPC server (GUI enabled).

### 4.3.2.1 Views

Views are the interfaces of the systems' client-side that are tasked with collecting information from the user, providing user feedback, and facilitating fundamental control over the software's functionality. The latter entails mechanisms for managing the database on the server side of the system, as well as the means to delineate the initiation and termination of the server-side system's active operation, including signal modification and traffic filtering.

Both (*graphical user interface*) (GUI) and *console* views are implemented, addressing the functional requirement FR9. The exact view to be used is determined by the presence of `--gui` argument at the application start-up.

**4.3.2.1.1 GUI View.** In order to ensure the user-oriented design, GUI was conceptualized through a multi-stage design process, beginning with low-fidelity (Lo-Fi) wireframes sketches that outlined the spatial arrangement of the interface elements, which provided a preliminary visual and interaction model (refer to Figures 4.6 and 4.7). Advancing to high-fidelity (Hi-Fi) wireframes, detailed mock-up of the GUI was created, incorporating aesthetic elements, such as color scheme (standard system theme is used) and typography, to yield a realistic user experience. These prototypes were then iteratively refined based on user feedback<sup>2</sup>, ensuring that the interface was intuitive, responsive, and aligned with user expectations. Final GUI representation corresponds to the latest Hi-Fi wireframe, and is denoted at Figures 4.8 and 4.9.

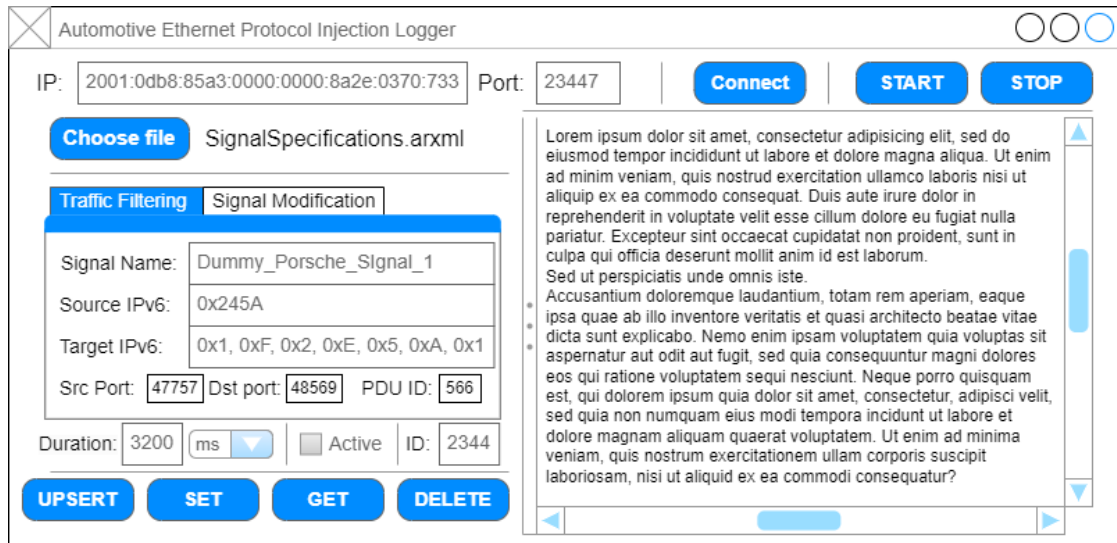


■ **Figure 4.6** Low-fidelity wireframe with the Signal Modification tab active.

The GUI view is represented by a *stateless* single Python class, which manages various widgets such as `QComboBox`, `QLineEdit`, and `QPushButton` (provided by PyQt6 library), which are systematically arranged within layouts. The resulting view employs both vertical and horizontal dividers to delineate distinct sections for 'server control', 'ARXML processing', 'traffic interference', 'common rule properties', 'database management', and 'feedback', ensuring a clear demarcation between input/output zones. A tab widget, which implements 'traffic interference' area, differentiates between 'Traffic Filtering' and 'Signal Modification' functionalities, enabling users to seamlessly toggle between these modes. Interactive elements like checkboxes, entry fields, and buttons are designed to capture user inputs for signal attributes, traffic filters and command

<sup>2</sup>The feedback was provided by the employees of Vehicle Testing Department at Porsche Engineering Services s.r.o. at the design stages.

execution, thereby streamlining the process of managing server-side database operations and controlling the overall functionality of a system.



■ **Figure 4.7** Low-fidelity wireframe with the Traffic Filtering tab active.

Despite meanings of particular GUI elements are quite intuitive, they are explicitly grouped depending on section and provided below:

#### Server Control:

- *Server IP:* An input field designated for the user to specify the Internet Protocol (IP) address of the target server; both IPv4 and IPv6 addresses are acceptable.
- *Server Port:* An input field serves for specifying the network port number on the target server for establishing a network connection.
- *Connect:* A control element (button) that servers for probing the client-server connectivity, referring to the **TEST** method defined in section 4.3.1.
- *START:* A control element (button) that switches the server operational mode, enabling the traffic interference (refer to **START** method defined in section 4.3.1).
- *STOP:* A control element (button) that switches the server operational mode, enabling the transparent gateway functionality (refer to **STOP** method defined in section 4.3.1).

#### ARXML Processing:

- *Choose file:* A control element (button) that initiates an **.arxml** file selection interface, enabling the user to select a configuration file required for the system's operation. After the file is chosen, it's name is displayed next to the button.

#### Traffic Interference – Signal Modification:

- *Signal Name:* The identifier of the signal intended for modification.
- *New Value:* A new value the specified signal has to be set to.
- *Polynomial:* 16 comma-separated hexadecimal polynomial coefficients used for in-PDU CRC calculation.
- *Secret Key:* 16 comma-separated hexadecimal cryptographic key partitions, used in the recomputation of MAC field.

**Traffic Interference – Traffic Filtering:**

- *Signal Name*: A packet, containing this signal, shall be marked for an exclusion from traffic (optional field).
- *Target IPv6*: Target (destination) IPv6 address of a packet to be excluded from traffic.
- *Source IPv6*: Source IPv6 address of a packet that has to be excluded from traffic.
- *Target Port*: Target (destination) port of a packet that has to be excluded from traffic.
- *Source Port*: Source port of a packet that has to be excluded from traffic.
- *PDU ID*: A packet, containing the specified PDU, shall be marked for an exclusion from traffic (optional field).

**Common Rule Properties:**

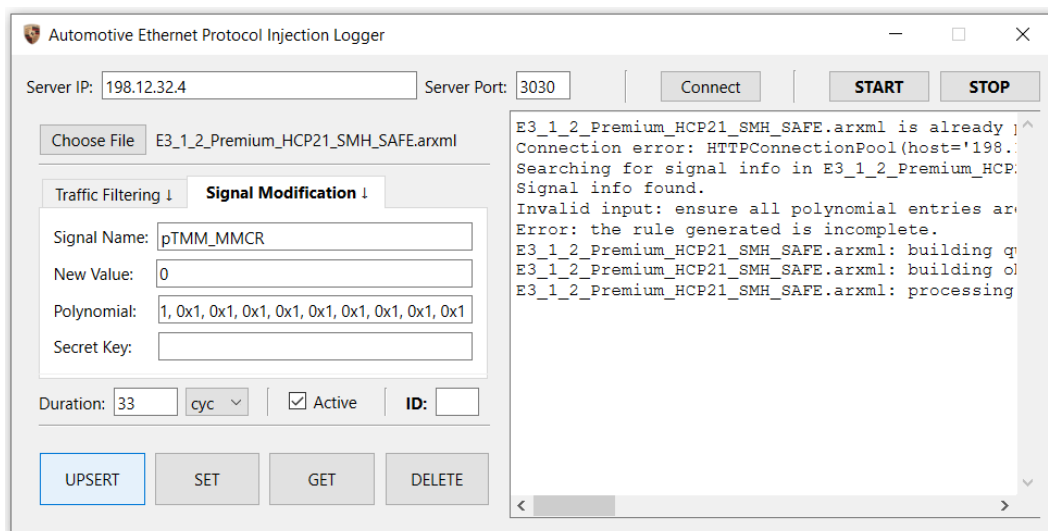
- *Duration*: Numerical representation of rule duration.
- *Duration Type*: Dropdown menu containing duration types: "ms", "s", "m", "cyc" or "inf".
- *Active*: Rule status upon insertion ('active' if checked, 'inactive' otherwise).

**Database Management:**

- *ID*: Rule identifier the database operation has to be performed with.
- *UPSERT*: Upsert a new rule into the database; (refer to UPSERT method in section 4.3.1).
- *SET*: Set the status of a rule to the specified one in the 'Active' input field; (refer to SET method in section 4.3.1).
- *GET*: Retrieve the specified rule from the database; (refer to GET method in section 4.3.1).
- *DELETE*: Permanently delete the specified rule from the database; (refer to DELETE method in section 4.3.1).

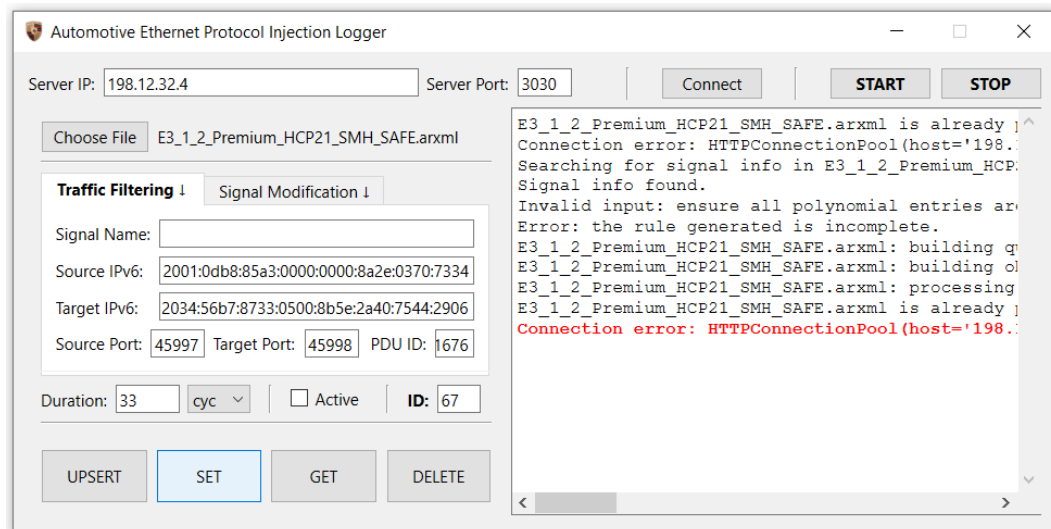
**Feedback:**

- System and user-targeted information, error logs, server responses are provided in the big scrollable (width is also adjustable) field, located on the right side of the program window.



■ **Figure 4.8** Client GUI at the time of operation with Signal Modification tab active.





■ **Figure 4.9** Client GUI at the time of operation with Traffic Filtering tab active.

**4.3.2.1.2 Console View.** In the Console View mode, the program is designed to execute a single operation dictated by the command-line arguments, provide feedback upon completion, and then terminate. Such a design facilitates the automation possibilities, making the program usage suitable for managing by different scripts. Since the command-line arguments are processed at the application start-up, the very Console View is responsible only for textual feedback outputting.

An example of program usage via Cosole View is provided below:

```
$ python3 app.py --server-ip 192.168.1.10 --server-port 8080 --method delete \
--rule-id 29
Server: No rule to DELETE with id 29 exists.
```

### 4.3.2.2 ARXML Parsing

ARXML files that describe inter-ECU communication may possibly differ slightly from one manufacturer to another, or depending on the specific ECU, making the very extraction non-trivial. The differences could include changes in organization of packages, naming conventions applied, etc. This implicitly suggests the design & implementation of a flexible mechanism for selective data extraction.

The final design decision is inspired with pre-existing proprietary developments in the field and breaks the AUTOSAR XML processing into several discrete steps, outlining the key abstract components participating in the parsing process as following:

- **Schema:** describes the data to be extracted and their hierarchy within ARXML file.
- **Parser:** analyzes user-defined schema and creates extraction queries; extracts the data specified by queries.

This approach is deemed to be flexible enough, as it allows for selective parsing of ARXML files, although demanding the knowledge of their internal organization. Moreover, it allows the future deployment of an ARXML parser as an independent software component, aligning with modular design and software reusability principles [65].

Overall, the established ARXML parsing and processing procedure is intended to be performed solely once per each ARXML file specified, saving all the extracted data into a separate folder.



**4.3.2.2.1 Schema.** The schema itself represents a collection of *objects* and their *values* with respect to their hierarchy. An object is defined as an aggregation of values with a given name and an anchor; an object can be either nested or not. A value represents the data to be extracted from the ARXML file. Each value is identified by a unique freely chosen name and includes parsing instructions, which referred later as queries.

The first object, identified by its anchor, marks the starting point for sequentially processing the following values. During the parsing process, each query uses the anchor of its parent object as a primary reference point. The parsing instructions are carried out later with respect to this anchor. It's important to note that an anchor can lead to either a single object, depending on the element's uniqueness in the ARXML file, or to an array of elements that meet certain criteria.

**4.3.2.2.1.1 Object Anchor.** The formulation of an anchor is critical as it delineates the precise location (*path*) within the XML hierarchy where the relevant elements are to be found. Additionally, object anchor itself supports several types, providing the higher flexibility in shema construction:

- **\_xpath:** This type of anchor allows the use of any XPath expression to define the object to be parsed.
- **\_ref:** This syntax is utilized for directly referencing a specific AUTOSAR object.
- **\_xref:** A hybrid of ‘\_xpath’ and ‘\_ref’, this anchor involves an XPath expression leading to an element containing an AUTOSAR Reference.

The ‘\_xref’ anchor is particularly noteworthy for its dual functionality. It is adept at facilitating the extraction of data from an element when only its AUTOSAR reference is accessible within the current context. A practical application of this is in retrieving the data type of a signal mapped to a PDU, where the PDU itself only contains a reference to the signal. The operational mechanism of ‘\_xref’ involves initially locating the element with the AUTOSAR reference, as specified by the XPath. Subsequently, it extracts the reference from the element’s text value and proceeds to resolve this reference, thereby establishing a new base for child value queries.

**4.3.2.2.1.2 Values.** A value’s path consists of an XPath expression that leads to the element where the data can be found. All types of XPath expressions can be used for path specification. Optionally, the path can be converted into an *inline reference* (if the referenced element is unique), by prepending `&(<xpath-to-ref>)` to the actual XPath expression:

value: `&(<xpath-to-ref>)]<xpath-to-element>`

Value queries can be further refined by adding attributes, specifying the extract location and format. This can be done by prepending additional parsing information (`>` and `:` symbols) to the path, as following:

value: `[location[>format]:]<xpath-to-element>`

Tables 4.8 and 4.9 denote the supported possible values for both extract location and format.

Syntax	Description	Usage
tag	Gets the tag of the element	value: tag:<xpath>
text	Gets the text of the element	value: text:<xpath>
@<name>	Gets the value of the specified attribute	value: @UUID:<xpath>

■ **Table 4.8** Value locations.

Syntax	Description	Usage
string	Takes the textual representation	value: text>string:<xpath>
int	Converts the value into an integer	value: text>int:<xpath>
float	Converts the value into float	value: text>float:<xpath>

■ **Table 4.9** Value formats.

**4.3.2.2.1.3 Schema Template.** Summarizing, the implemented schema markup provides means for flexible and selective specification of data to be extracted from ARXML file. The primary reason for developing this schema notation is to align the intrinsic structure of AUTOSAR XML files with the hierarchical organization of objects and values' names defined in the schema template<sup>3</sup>, as indicated in Listing 4.1.

■ **Code listing 4.1** Schema in .yaml format used for ARXML parsing.

```
Frames:
  _xpath: ".//SOCKET-CONNECTION-BUNDLE"
  Source: "SHORT-NAME"
  SocketConnection:
    _xpath: ".//SOCKET-CONNECTION"
    DestinationContainer:
      _xref: "CLIENT-PORT-REF"
      Destination: "SHORT-NAME"
    Pdus:
      _xpath: ".//PDUS/SOCKET-CONNECTION-IPDU-IDENTIFIER"
      Id: "HEADER-ID"
      PropertiesContainer:
        _xref: "PDU-TRIGGERING-REF"
      Properties:
        _xref: "I-PDU-REF"
      Signals:
        _xpath: ".//I-SIGNAL-TO-I-PDU-MAPPING"
        Name: "SHORT-NAME"
        StartPosition: "text>int:START-POSITION"
        Length: "text>int:\&(amp;I-SIGNAL-REF)LENGTH"
```

In particular, this schema marks the following values for extraction, being later placed into a nested 'Frame' object:

1. *Source*: source port & IPv6 address of an Ethernet packet.
2. *Destination*: destination port & IPv6 address of an Ethernet packet.
3. *Id*: PDU ID.
4. *Name*: signal name.
5. *StartPosition*: absolute offset in bits from the beginning of the SDU within PDU, indicating where the signal starts.
6. *Length*: absolute signal length in bits.

► **Note 4.4.** The selection of the YAML format for the configuration schema was a deliberate choice, aimed at achieving a logical segregation of the various types of data associated with the software system: YAML for configuration settings, ARXML for input, and JSON for output.

<sup>3</sup>This schema template is chosen, since it reflects the *plausible* structure of real ARXML files used by automotive companies, the resulting software is intended to operate with.

It is noteworthy to mention, that the selected schema-based approach for ARXML parsing is beneficial, generally resulting in possibilities of extraction of any data, according to schema definition.

**4.3.2.2.2 Parser.** The 'Parser' (or 'ARXML Parser') abstract component of AEPRIL-client represents a collection submodules targeted at schema analysis and the very ARXML processing. The schema analysis, in turn, can be divided into *building of queries* and *handling of queries*, where a *query* represents a set of instructions, describing the ARXML parsing process.

Building of queries implies a proper treatment of schema-defined structures, which is established by utilization of adequate containers for internal data representation (see paragraph 4.3.2.2.2.1 for further details). On the other hand, handling of queries involves:

- **Path Handling:** treatment of object's anchor in order to identify it within an ARXML file (refer to 4.3.2.2.2.3).
- **Value Handling:** treatment of individual values marked for extraction (refer to 4.3.2.2.2.4).
- **Object Handling:** treatment of data marked for an extraction from hierarchical point of view (refer to 4.3.2.2.2.5).

All the query-related functionality (both building and handling) is detailed in paragraph 4.3.2.2.2.6. Additionally, the ARXML processing itself is generally done by construction and execution of XPath expressions according to the queries built, at the time of their handling. Detailed information on functionality related to ARXML processing is denoted in paragraph 4.3.2.2.2.2. Details about YAML schema uploading and saving of extracted data are described in paragraphs 4.3.2.2.3 and 4.3.2.2.4 respectively.

**4.3.2.2.2.1 Internal Data Representation.** User-defined YAML schema, denoted in paragraph 4.3.2.2.1, represents a collection of data that has to be stored in the proper format for further processing. This is performed by storing the information provided in a schema into instances of three data structures (classes):

- **DataValue** represents a single data item, associated with a query that describes how to extract or interpret the value.
- **DataQuery** represents a container, encompassing the path of data extraction, the specific value to be retrieved and coherent attributes.
- **DataObject** represents a structured entity, encapsulating a named collection of data values and/or nested data objects, linked with a specific path.

**4.3.2.2.2.2 ARXML Processing.** The **Parser** class encapsulates critical functionalities for interfacing with AUTOSAR XML files directly. Upon initialization, the class employs the `lxml` library's `XMLParser`, which is configured to discard blank text, thus enhancing parsing efficiency. The parsed XML tree and its root are stored as private attributes, ensuring encapsulation.

**Parser** performs dynamic extraction and utilization of XML namespaces; this is achieved through the `nsmmap` property of the root element.

The class method `find_all_elements(...)`<sup>4</sup> exemplifies the application of XPath expressions, tailored for AUTOSAR's XML schema. It leverages the class's capability to dynamically assemble XPath queries, thereby providing a versatile tool for element retrieval based on varied path inputs.

---

<sup>4</sup>Hereinafter, the `function(...)` notation will be used to name specific methods; the `...` symbols mean the presence of function arguments.

In the realm of reference resolution, the `find_reference(...)` method showcases a traversal logic, navigating through the nested structure of ARXML files & dissecting AUTOSAR reference into its constituent parts and iteratively resolving it against the XML hierarchy.

Auxiliary methods such as `find_element_by_shortcode(...)` and `get_shortcode(...)` are tailored for AUTOSAR's peculiarities, like the pervasive use of 'SHORT-NAME' tags, focusing on common patterns and requirements in ARXML file handling.

**4.3.2.2.2.3 Path Handling.** The `PathHandler` class is specifically designed for retrieving XML elements based on different types of paths defined in `DataQuery`. The constructor of the class initializes it with an `Parser` instance, enabling the use of specialized parsing functions.

The key method, `elements_by_path(...)`, handles two types of paths:

- **DataQuery.XPath:** Specified elements are retrieved using `elements_by_xpath(...)` function. If the path is an inline reference, the method ensures that exactly one reference (refer to paragraph 4.3.2.2.1.2) is found, raising an exception otherwise.
- **DataQuery.Reference:** Corresponding element is directly retrieved via execution of the `element_by_ref(...)` method.

The `elements_by_xpath(...)` and `element_by_xpath(...)` methods facilitate the retrieval of elements using XPath expressions. The former returns a list of elements, while the latter retrieves a single element. These methods rely on the `Parser`'s capability to assemble and execute XPath queries.

To be precise, the `element_by_ref(...)` method is designed to fetch elements based on their reference ID, a common requirement in ARXML file parsing. The `element_by_inline_ref(...)` method introduces specialized handling for inline references, splitting the inline reference into two parts: a reference path and a value path, using the `__split(...)` method. Additionally, the method includes a performance optimization for 'SHORT-NAME' values, where it returns the *referencing* element directly instead of the *referenced* one.

**4.3.2.2.2.4 Value Handling.** Value handling is being managed by classless submodule `value_handler`. The functions `handle(...)`, `__get_value(...)`, and `__convert_value(...)` are integral to the ARXML parsing process, primarily focusing on extracting & transforming values from XML nodes based on specified queries. The `handle(...)` function serves as the entry point, determining the value extraction strategy based on the type and characteristics of the provided `DataQuery` object. Notably, it includes a specialized treatment for inline references in `DataQuery.XPath` objects, specifically targeting `SHORT-NAME` references<sup>5</sup>.

In the extraction phase, the `__get_value(...)` function is utilized to retrieve the value from the XML node, being capable of handling different value types denoted in Table 4.8. The attribute retrieval, denoted by a prefix '@', involves a check for the attribute's existence, ensuring robust error handling and logging of missing attributes, essential for debugging and data integrity.

The transformation phase is handled by `__convert_value(...)`, where the extracted value is converted according to the specified format in the `DataQuery` object, the function is operating with. This function supports formats denoted in Table 4.9.

**4.3.2.2.2.5 Object Handling.** The `ObjectHandler` class is a key component in the AUTOSAR XML parsing framework, designed for processing `DataObject` and `DataValue` instances. This class is essential for interpreting the hierarchical structure of ARXML files and extracting relevant data based on specified queries. It initializes with a worker object (refer to 4.3.2.5.3) for progress reporting and an instance of `Parser`, encapsulated within `PathHandler`, to handle path-based queries.

---

<sup>5</sup>*SHORT-NAME* is a key attribute of ARXML notation which serves for identification of nodes by name.

The primary method, `handle(...)`, orchestrates the data extraction process. It begins by identifying the root node of the XML structure if no specific node is provided, marking the entry point for data processing. The very method then retrieves XML elements corresponding to the path specified in the given `DataObject` using `PathHandler`. For each element found, it iterates and processes nested `DataValue` and `DataObject` instances through the `__handle_values` method.

The `__handle_values(...)` method is primarily responsible for recursive processing of nested `DataObject` and `DataValue` instances:

- For `DataObject` instances, it recursively calls the `handle(...)` method.
- For `DataValue` instances, it utilizes `__handle_value(...)` to extract and convert data based on the specified query and format.

This recursive approach enables the handling of complex, nested data structures (common characteristic of ARXML), as denoted in user-defined YAML schema.

The `__handle_value` method focuses on processing individual `DataValue` instances. It differentiates between XPath and reference-based queries, utilizing the `PathHandler` for retrieving the corresponding XML element. Once the element is obtained, the method delegates the value extraction and conversion to the `value_handler` submodule, which handles the specifics of data conversion based on the query format.

**4.3.2.2.2.6 Query Handling.** First, the `QueryBuilder` class is pivotal in constructing queries from a schema-defined dictionary. Its primary function, `build(...)`, iterates over the schema entities, transforming each key-value pair into a `DataObject` instance. Further key methods of this class are provided as pseudo code in Algorithms 1, 2 and include:

- `__parse_object(...)`: This private method is integral to the query-building process. For each configuration item, it determines the type of path (`XPath`, `xref`, or `ref`) and constructs the corresponding `DataQuery` object. It also recursively processes nested dictionaries, allowing for hierarchical data structures.
- `__parse_value(...)`: This method handles individual data values by parsing the given value string into a `DataQuery`, which defines how to retrieve the value from an ARXML file.
- `__parse_query(...)`: This method is tasked with parsing query strings into `DataQuery` objects, extracting and interpreting the path, value, and format components.

Auxiliary methods such as `__get_path(...)`, `__get_value(...)`, and `__get_format(...)` are used to parse specific components of a query, ensuring correctness and consistency in the generated `DataQuery` objects.

Additionally, the `QueryHandler` class is responsible for executing the queries against an ARXML file. Key functionalities encapsulated into `handle_queries(...)` method include:

- **File Validation:** The method ensures that the input is a valid ARXML file, checking for its existence, file type, and appropriate extension.
- **Data Processing:** Upon validation, the method processes each `DataObject` by interfacing with an `ObjectHandler`. The `ObjectHandler` utilizes the `Parser` to navigate the ARXML file structure and extract or compute the values as defined by the queries in `DataObjects` (see paragraph 4.3.2.2.2.5).
- **Result Aggregation:** The processed results are aggregated into a dictionary, linking each `DataObject`'s name to its corresponding outcome.

**Algorithm 1** Parse Object

---

```

1: function PARSEOBJECT(name, values)
2:   required  $\leftarrow$  {'_xpath', '_xref', '_ref'}
3:   pathValue  $\leftarrow$  required  $\cap$  KEYS(values)
4:   if LENGTH(pathValue)  $\neq$  1 then
5:     RAISEERROR("Missing anchor")
6:   end if
7:   path  $\leftarrow$  DETERMINEPATH(pathValue, values)
8:   dataValues  $\leftarrow$  []
9:   for (key, value)  $\in$  values do
10:    if key  $\in$  required then
11:      continue
12:    end if
13:    if ISDICT(value) then
14:      dataObject  $\leftarrow$  PARSEOBJECT(key, value)
15:      APPEND(dataValues, dataObject)
16:    else
17:      dataValue  $\leftarrow$  PARSEVALUE(key, value)
18:      APPEND(dataValues, dataValue)
19:    end if
20:  end for
21:  return CREATEDATAOBJECT(name, path, dataValues)
22: end function

```

---

**Algorithm 2** Parse Value and Query

---

```

1: function PARSEVALUE(name, value)
2:   query  $\leftarrow$  PARSEQUERY(value)
3:   return CREATEDATAVALUE(name, query)
4: end function
5: function PARSEQUERY(text)
6:   if not CONTAINS(text, pathSeparator) then
7:     path  $\leftarrow$  GETPATH(text)
8:     return CREATEDATAQUERY(path)
9:   end if
10:  [rawValueFormat, rawPath]  $\leftarrow$  SPLIT(text, pathSeparator)
11:  path  $\leftarrow$  GETPATH(rawPath)
12:  if CONTAINS(rawValueFormat, formatSeparator) then
13:    [rawValue, rawFormat]  $\leftarrow$  SPLIT(rawValueFormat, formatSeparator)
14:    value  $\leftarrow$  GETVALUE(rawValue)
15:    format  $\leftarrow$  GETFORMAT(rawFormat)
16:  else
17:    value  $\leftarrow$  GETVALUE(rawValueFormat)
18:    format  $\leftarrow$  StringFormat
19:  end if
20:  return CREATEDATAQUERY(path, value, format)
21: end function

```

---

**4.3.2.2.3 Schema Uploading.** The `ConfigProvider` class, is quite simple and stands as a cornerstone for loading and parsing configuration data. This class is characterized by two

principal methods: `load(...)` and `parse(...)`.

The `load(...)` method is tailored for reading the user-defined YAML schema. Before loading the file, the method employs a validation process to ensure the file extension is `.yaml`. Upon successful validation, the method proceeds to read the YAML file using the `safe_load(...)` function from the `yaml` module.

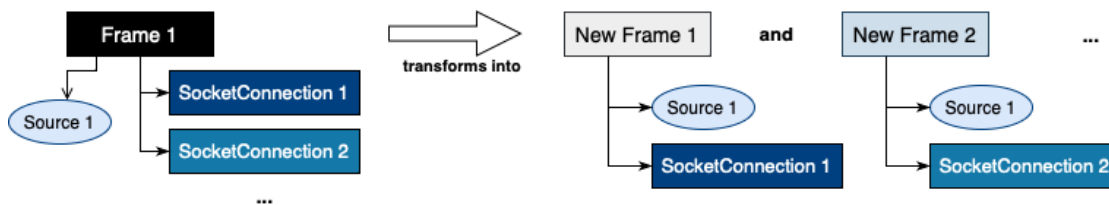
The `parse(...)` method, on the other hand, extends the class's functionality by allowing the parsing of YAML content directly from a string, also employing the `safe_load(...)` function.

**4.3.2.2.4 Saving of Extracted Data.** The `DataWriter` class, intricately designed within the system, plays a pivotal role in writing and transforming data extracted from ARXML file. It comprises two primary methods: `transform_data(...)` and `write_json(...)`.

Particularly notable is the `transform_data(...)` method's handling of potential null values and non-list structures, a common challenge in data transformation tasks[66]. It iteratively processes each 'Frame' within the input data, systematically ensuring that components such as 'SocketConnection', 'Pdus', and 'Signals' (refer to 4.3.2.2.1.3) conform to list structures.

► Note 4.5. Common practices of ARXML structuring include *one-to-many* mapping of elements. For instance, one 'Frame' could potentially have one or many 'SocketConnection' elements attached to it, resulting into nested tree structure. This uncertainty is resolved by an explicit conversion to list, ensuring that all elements are treated in the same way.

Since the thesis is focused on manipulation with signals, 'Pdus' objects having no 'Signals' objects attached ('SocketConnection' objects having no 'Pdus' attached, respectively) are excluded by the method from the extracted data. Moreover, in order to simplify the one-to-many mapping of 'Frame' and 'SocketConnection' objects, the method performs explicit *one-to-one* mapping by multiple construction of new 'Frame' objects having only one 'SocketConnection' attached, preserving essential data while restructuring it into a more accessible and consistent format (refer to Figure 4.10).



■ **Figure 4.10** Transformation of one old 'Frame' object into two new 'Frame' objects.

After the transformation was performed, the resulted data are easily readable and could be used far beyond the scope of this thesis (testing, calibration, etc.) The *plausible* layout of extracted and transformed data is provided in Listing 4.2 below:

■ **Code listing 4.2** Generated frame specification in JSON format.

```
{
  "Source": "SAB_SA_Tx_fd53_7ab8_0383_0005_0000_0000_0000_0010_42_UDP",
  "Destination": "SH_Rx_fa31_6ac2_0000_0000_0000_0000_0000_0005_45_UDP",
  "Pdus": [
    {
      "Id": "2101",
      "Signals": [
        {"Name": "Eth_Gateway_Config", "StartPosition": 0, "Length": 2},
        {"Name": "Cmn_Transport_Mode", "StartPosition": 2, "Length": 4}
      ]
    }
  ]
}
```



```

    {
      "Id": "290601",
      "Signals": [
        {"Name": "pTMM_MMTR", "StartPosition": 0, "Length": 1},
        {"Name": "pTMM_MMCR", "StartPosition": 1, "Length": 1}
      ]
    }
  ]
}

```

The `write_json(...)` method extends the class's functionality by writing the transformed data into `.json` files. This, in turn includes the creation of one formatted JSON file containing all the frames, and multiple indexed files with one-line formatting for each frame separately (all inside the same folder). The following example of `dir` command output in Windows command line showcases the naming convention<sup>6</sup> applied to generated files:

```

...
1/20/2023  04:47          139,365 E3_1_2_Premium_ADAS_DC.json
11/20/2023 04:47           3,096 E3_1_2_Premium_ADAS_DC_frame_1.json
11/20/2023 04:47           2,109 E3_1_2_Premium_ADAS_DC_frame_2.json
11/20/2023 04:47           2,092 E3_1_2_Premium_ADAS_DC_frame_3.json
...

```

### 4.3.2.3 Rule Creator

The `RuleCreator` class plays a pivotal role in creation of rules and extracting signal information from prepared JSON frame specifications.

The creation of the very rules depends on the operational mode inherent to rule itself. If the desired configuration implies filtering of packets, the `Rule` class instance is being created according, to the user-provided data, and returned upon `create_rule(...)` method invocation, setting up the network rule with various parameters such as source and destination IPs, ports, and other relevant information. Otherwise, the rule is being completed with utilization of `generate_specs(...)` method.

► Note 4.6. Importantly, if the signal name is provided in the traffic filtering mode, the rule is being created similarly to the signal modification rules.

This method iterates over JSON files containing frame data, systematically searching for the required signal (DFS), updates the rule object with detailed signal information, and calculates progress, enhancing the user experience with real-time feedback.

Furthermore, the `find_signal_info(...)` method, used in this process, serves for exact parsing and extraction of signal-related data, such as start position, length, etc. After the signal is found, the associated CRC, BZ, and MAC information is being searched within the same PDU and added to rule if found, therefore identifying the security mechanisms applied to a particular PDU.

► Note 4.7. The presence of security mechanisms for a specific PDU is determined by the presence of signals, which names end with `'_BZ'`, `'_CRC'` and `'_MAC'` respectively.

Additionally, as it is denoted in listing 4.2, both `'source'` and `'destination'` fields in the frame specification are complex and include IPv6 address, port, and protocol type. The auxiliary function `extract_ipv6.port.protocol(...)` is designed to dissect these fields, separating those data into distinct properties of a rule.

<sup>6</sup>*E3.1.2-Premium-ADAS-DC* is a name of processed ARXML file containing the frames specifications.



#### 4.3.2.4 Controller

The `Controller` class is central to the system; it collaborates closely with the `Model` and view classes (`GuiView` and `ConsoleView`), thereby segregating the system's data handling, business logic, and user interface, integrating into the MVC design pattern. The primary concern of this class is to perform data handling (validation, transformation) from user to Model and vice versa.

The class initializes with parameters that determine the type of user interface (GUI or console) and the option to enable logging. The core of the class is the `generate_request(...)` method, which serves as the primary mechanism for processing JSON-RPC requests denoted in section 4.3.1, each requiring specific parameters and validation checks. The method includes sophisticated error handling and user feedback mechanisms, informing user of potential issues coherent with request parameters.

Additionally, the class comprises methods for setting the server information and triggering the processing of ARXML files.

#### 4.3.2.5 Model

The `Model` class is an essential element of a client-side of a system; it represents the business logic of an application and designed to manage network communications and file operations in particular. This class interfaces with a controller and uses a worker-thread model (refer to paragraph 4.3.2.5.3) for executing tasks in a non-blocking manner, particularly beneficial in graphical user interface (GUI) settings.

Central to the class's functionality is the `process_request(...)` method, which triggers the rule creation process and handles JSON-RPC requests specified in section 4.3.1. In GUI contexts, it employs a `QThread` to delegate processing tasks to a `Worker` object, ensuring that the GUI remains responsive during operations.

Another key feature is the `process_arxml(...)` method, which processes AUTOSAR XML files according to YAML schema using a similar worker-thread approach.

The `check_rule(...)` method is crucial for ensuring the compliance of user-provided data with rules generated by the system. It checks for essential elements like 'Secret Key' and 'Polynomial' in user provided data, which are necessarily have to be provided when PDU is protected by CRC or AUTOSAR SecOC.

**4.3.2.5.1 JSON-RPC Client.** The `send_request_to_server(...)` method inherent to the class is responsible for facilitating server communication, and consequently, implementing the JSON-RPC client abstract component denoted at Figure 4.1. It uses IP address validation (IPv4 and IPv6 addresses) and HTTP request handling to interact with the server, incorporating comprehensive error management to address potential network issues. The method is executed not asynchronously (on the same thread as a GUI, if enabled).

**4.3.2.5.2 Rule Representation.** Within the client-side of a system, the rule is being stored into `Rule` class instance. This class represents a functionless container dedicated exclusively for the storage of information. The member variables of a class directly reflect the rule specification aforementioned at Note 4.1.

**4.3.2.5.3 Worker Object.** The `Worker` class, leveraging the PyQt6 framework, serves as a crucial but auxiliary component within the system's architecture. This class is instantiated with a `model` object, which carries the necessary configurations and state for the workflow.

It encapsulates a series of operations associated primarily with the data processing, and is dedicated to perform these operations on a separate thread asynchronously (if GUI enabled). The sequence of these operations – loading configuration, building queries, extracting data, writing data, and creating rules – is being orchestrated by the class itself. The class employs a series of

collaborator objects, namely `ConfigProvider`, `QueryBuilder`, `QueryHandler`, `DataWriter`, and `RuleCreator`, to execute aforementioned tasks.

The class also utilizes PyQt6's signal-slot mechanism [67], exemplified by signals such as `progress`, `message`, `error`, `exception`, and `rule`, to communicate the status and results of its operations, thus enabling real-time feedback.

Exception handling is a notable aspect of this class, with a dedicated `emitException(...)` method designed to emit detailed exception messages. This method enhances the robustness of the system by providing comprehensive diagnostic information to user in case of failures.

### 4.3.3 Server

Initially, the development of the server-side of the system is primarily aimed at the implementation of requirements falling under the 'Action' category as previously acquired. This, in turn, encompasses the appropriate processing of data received from the client, ensuring their accurate storage and interpretation. Furthermore, the server component of the system must, when necessary, execute direct intervention in network traffic in accordance with the established rules. Ensurance of minimal overhead at the time of intervention is identified as a paramount concern, addressing the demands on compliance with ASIL FTTI (see section 2.1.2).

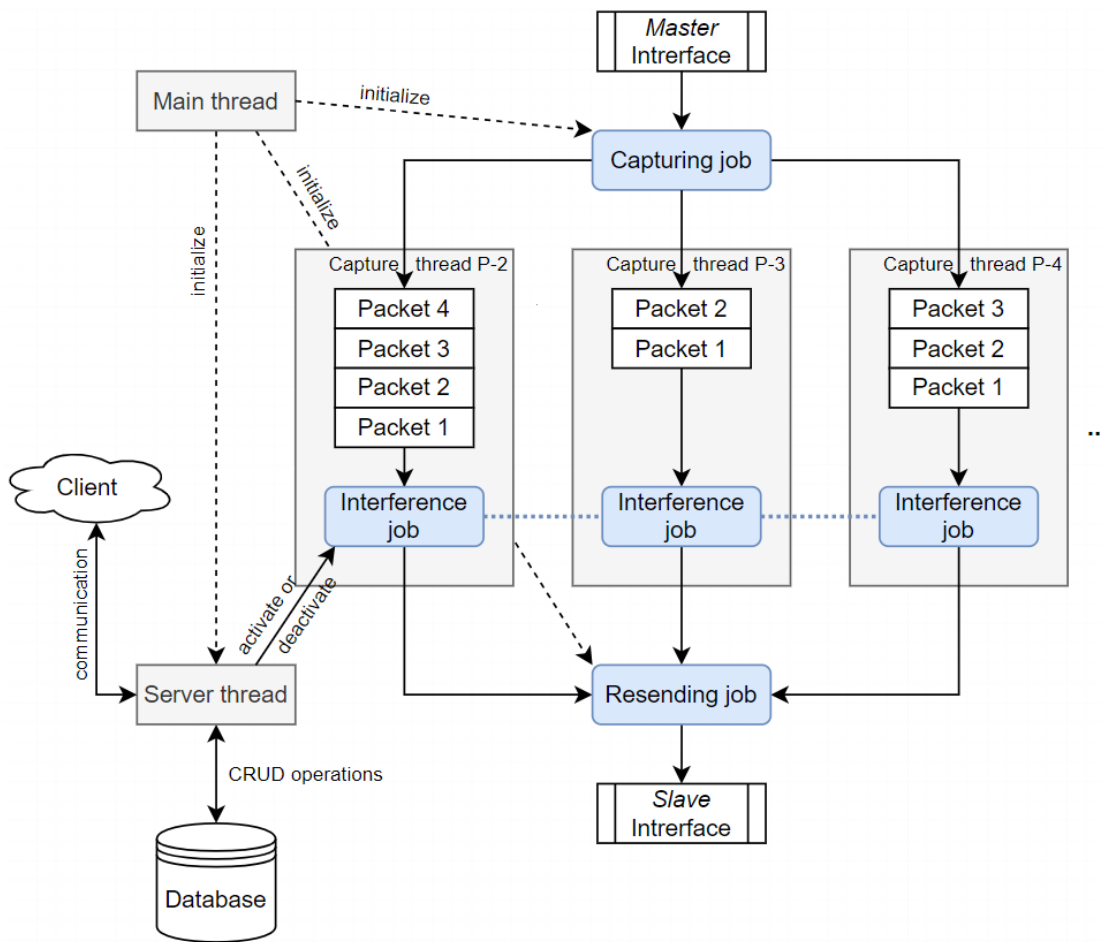
Each of the aforementioned responsibilities is assigned to a specific component within the server's architecture, as delineated in Figure 4.1. This allocation implies that each component may be composed of either none or several C++ classes. The internal mechanics of the software operation are schematically depicted in Figure 4.11. Additionally, this figure illustrates that the server-side of the system operates across multiple distinct threads:

- **Main application thread** is utilized for performing all the activities needed at the start-up of the application as well as for *'keeping the application alive'* at the time of system execution. This thread is mainly used by `SystemLogic` abstract component (refer to section 4.3.3.3).
- **JSON-RPC server thread** is utilized for real-time listening of the server on specified port and is dedicated to `JSON-RPC Server` component (refer to section 4.3.3.7). Moreover, database operations (refer to section 4.3.3.5) are executed at this thread in synchronous manner as well.
- **Packet capture threads** are utilized for real-time capture of packets and their processing; number of those threads is bounded to  $P - 2$  at most, where  $P$  represents the number of physical cores of the processor. Those threads are being used independently from each other by `PacketParser` abstract component, denoted in section 4.3.3.6.
- **Rule deactivation threads** are *optional* and sleeping threads, belonging to no discrete component; they are used to control the deactivation of a rule according to the specified duration of its application (refer to paragraph 4.3.3.8.1). Number of those threads corresponds to the number of currently active rules marked for delayed deactivation, where the precise delay is provided in time units.

The software utilizes conditional compilation to determine the availability of particular features during the program lifecycle. Conditional compilation, as well as the adjustment of particular hard-coded constraints, is managed by macros defined in `customizable conf.h` header file. In addition, server-side of a system does not support any GUI; the initial settings for the software are passed via command line arguments and cannot be changed at the runtime.

► **Note 4.8.** The detailed information on command line arguments is provided to user when `--help` argument is specified.

Moreover, since the client-server interface established in section 4.3.1 does not support methods intended to completely shut down the server side of a system, it only can be performed manually (`Ctrl+C`).

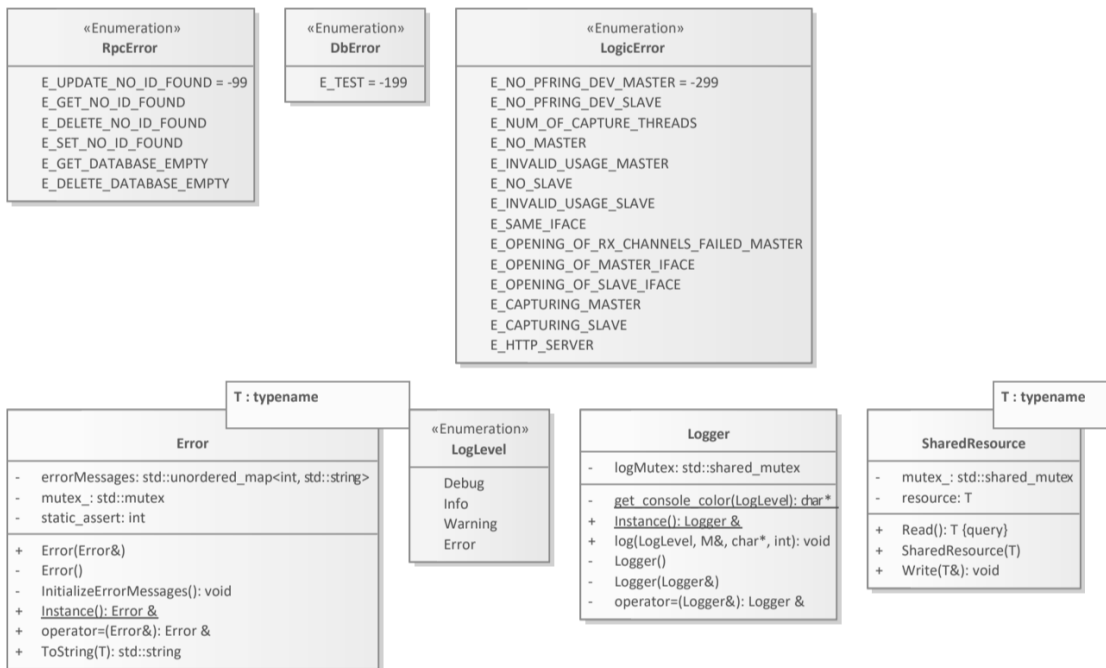


■ **Figure 4.11** Internal principle of the system's server-side.

► **Note 4.9.** The *interference job* depicted at Figure 4.11 is *iteratively* applied to each of packet in a bunch of incoming packets, if the server side of a system is not in the transparent gateway mode (otherwise this job is simply skipped). In context of signal modification, it implies searching for a specific PDU within a packet payload and substituting signal bits with required value, also performing analogous substitution for bits associated with present security mechanisms. Conversely, in terms of traffic filtering, it merely implies the exclusion of specified packet from traffic.

#### 4.3.3.1 Common

Apart from implementation components, responsible for direct addressing of functional software requirements, further functionalities that are shared among all system modules, such as robust error handling, comprehensive logging, and synchronized resource management have to be implemented as well. The precise specifications of classes and enumerations coherent with resolving of those issues are provided at Figure 4.12.



■ **Figure 4.12** Specifications of 'common' classes and enumerations.<sup>7</sup>

**4.3.3.1.1 Logging.** The `EventLogger` class encapsulates a thread-safe logging mechanism and is implemented as a Singleton to ensure that a single, global logging instance is utilized throughout the system. The class constructor is private, and both the copy constructor and assignment operator are deleted, reinforcing the Singleton pattern and preventing multiple instantiations. The static method `Instance()` guarantees access to the singular `Logger` instance.

The `log` function template is a critical member of this class, accepting a log level, a message, and the originating file and line number, facilitating contextual output. It leverages the `std::chrono` library to timestamp log entries with high precision and employs `std::strftime` to format these timestamps into a human-readable string. The `LogLevel` enumeration dictates the granularity of the logs, with *Debug*, *Info*, *Warning*, and *Error* levels corresponding to increasing levels of severity.

Depending on the compilation flags `CONSOLE_LOGGING` and `FILE_LOGGING`, the log messages are conditionally directed to the console with appropriate color coding, and/or appended to a log file, respectively. This is achieved within a scope guarded by a `std::scoped_lock` on `logMutex`, ensuring mutual exclusion in multithreaded scenarios.

Additionally, macros such as `LOG_DEBUG`, `LOG_INFO`, `LOG_WARNING`, and `LOG_ERROR` are defined to simplify the invocation of the `log` function and can be enabled or disabled via preprocessor directives, allowing for flexible control over logging verbosity during different stages of development or during the intended usage.

**4.3.3.1.2 Error handling.** Within the `err` namespace, the error handling infrastructure is meticulously designed to categorize and manage potential errors within the system. First, the macro `OK` set to 0, serves as the indicator of successful operations; it is used as anticipated return value for some functions of a program. Three distinct enumerations, `RpcError`, `DbError`, and `LogicError`, are declared to represent and intuitively separate various error states specific to

<sup>7</sup>This figure, as well further similar presented later, was automatically generated with Sparx Systems Enterprise Architect based on the *available* source code.

remote procedure calls, database operations, and logical processing, respectively. These enumerations encapsulate error codes, uniquely identifying each cause of a plausible error.

The `Error` class template, parametrized by an enumeration type `T`, ensures that only enumeration types are permissible as template arguments, enforced by a `static_assert`. The class follows the Singleton design pattern, providing a thread-safe, globally accessible instance via the `Instance()` method. This design choice guarantees a single, coherent point of error reporting throughout the application's lifecycle. The method `ToString(T code)` converts enumeration values to human-readable messages, secured by mutex locks to prevent race conditions in multi-threaded contexts.

Initialization of the error messages is conditionally compiled using the `if constexpr` construct, allowing the class to be tailored for specific error categories during the compilation. Error messages are stored in `std::unordered_map` container, associating integer error codes with corresponding descriptive strings. The class's constructor remains private, reinforcing the Singleton pattern and invoking `InitializeErrorMessages()` to populate the error messages relevant to the instantiated type. Notably, the class prohibits copy construction and assignment to maintain the integrity of the Singleton instance.

**4.3.3.1.3 Shared Resource.** The `SharedResource` class template lies within `utils` namespace and encapsulates thread-safe operations for managing shared data. Conceived to operate in a concurrent environment, it employs `std::shared_mutex` to coordinate access to a shared resource of the generic type `T`. The constructor `SharedResource(T initialValue = T())` initializes the resource with a default or specified value. The member function `void Write(const T &newValue)` employs `std::unique_lock` to ensure exclusive access for writing operations, thus updating the resource without interference. Conversely, the `T Read() const` function uses `std::shared_lock`, allowing multiple concurrent read accesses while maintaining data integrity.

This template is explicitly instantiated for types used, such as `bool`, `uint8_t`, and `int32_t`.

**4.3.3.1.4 Utility Functions.** The utility functions are identified as functions, that are not logically connected with a specific existing class. Those include functions within `utils` namespace, intended for data transformation and time conversion.

The `serializeVector(...)` function abstracts a byte array into a comma-delimited string, while its reciprocal, `deserializeVector(...)`, parses a string back into a byte array, facilitating data reconstruction. These two functions are primarily intended to handle 'Polynomial' and 'Secret Key' fields of a rule before importing it to or after exporting it from the database. The `ConvertToDuration(...)` function, interprets numerical values paired with temporal units ('ms', 's', 'm'), translating them into a `std::chrono::milliseconds` object, while incorporating a predefined offset to addresses possible processing delay (equal to 1 millisecond by default).

### 4.3.3.2 I/O Interfaces

In context of traffic processing, both input (*master*) and output (*slave*) interfaces are being represented by the `pcpp::PfRingDevice` class, which stands as an advanced abstraction of the `PF_RING` library. It supports concurrent operations across multiple network interfaces, enabling the simultaneous usage of various network channels.

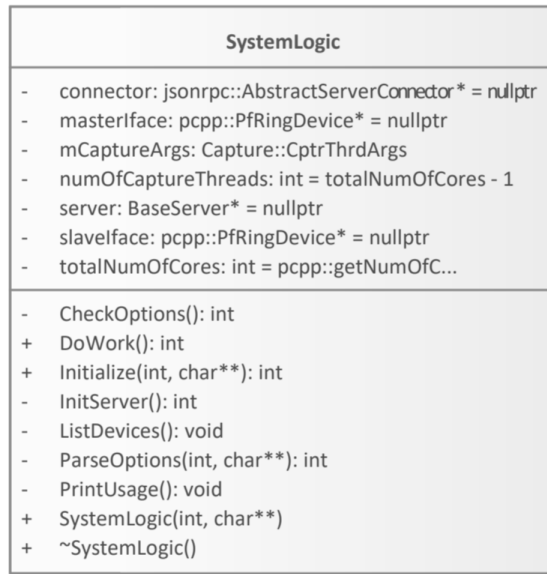
Despite Automotive Ethernet is a full-duplex communication bus, the program design assumes the usage of separate interfaces for both packet capturing and resending respectively. This implies the necessity to have two instances of the very class initialized as member variables of `SystemLogic` class (see section 4.3.3.3).

In more detail, the initialization process is performed for each interface via execution of `pcpp::PfRingDeviceList::getInstance().getPfRingDeviceByName(ifaceName)`, where the function parameter represents the identifier of the desired network interface available within the

system. The role of master interface within the entire program is bounded solely to concurrent capturing of packets via `startCaptureMultiThread(...)` function, while slave interface is limited only to sending the processed packets via `sendPackets(...)` function call.

### 4.3.3.3 System Logic

The `SystemLogic` class is a critical component of the system, orchestrating the parsing command-line arguments, initialization & management of network packet capturing and JSON-RPC server. It's specification is available at Figure 4.13.



■ **Figure 4.13** Specification of `SystemLogic` class.

**4.3.3.3.1 Command-Line Arguments.** `ParseOptions(...)` method is executed in the class' constructor. It interprets command-line inputs to dynamically configure network interfaces and number capture threads. Utilizing `getopt_long(...)` of `<getopt.h>` header file, the function parses options such as interface names and thread counts, ensuring the robust setup of network parameters. Upon completion of the argument parsing process, the parsed arguments undergo a verification procedure for logical consistency using the `CheckOptions()` method; this validation step is also embedded within the constructor of the class.

The error handling within named methods ensures that any misconfiguration, missing arguments or their incorrect usage result in appropriate logging and system exit.

**4.3.3.3.2 Initialization.** In the `Initialize(...)` method, the class initializes the `Pcap-PlusPlus` library with `pcpp::AppName::init(...)` and sets up a JSON-RPC server on a separate thread using `InitServer(...)`. The server setup involves instantiating `BaseServer` and `jsonrpc::HttpServer`, followed by an attempt to start the server with error handling for potential startup failures. The server is configured to listen on a designated port, denoted in `conf.h`, with JSON-RPC 2.0 specifications.

Additionally, this method prepares the `PF_RING` devices for packet capturing in multi-thread mode. It configures the master interface for receiving data, ensuring per-flow packet distribution among threads, while the slave interface is set up for transmission using the `open()` method. Extensive error logging is employed using the `LOG_ERROR` macro, referencing specific `LogicError` codes.

**4.3.3.3.3 Execution.** The `DoWork()` method is pivotal for the system's runtime operation. It initializes a boolean flag `shouldStop` to control the execution loop and registers a callback `onApplicationInterrupted` to handle application termination. This method configures packet capture settings, calculating the core mask and starting multi-threaded capturing on the master interface. It then enters an infinite loop, periodically checking the `shouldStop` flag, thus maintaining the system operational until an interrupt signal is received.

**4.3.3.3.4 Clean-up.** The class' destructor emphasizes resource management, where network interfaces are stopped for packet capturing and closed, and server resources are deallocated to ensure no memory leaks upon system exit.

**4.3.3.3.5 Auxiliary Functions.** `PrintUsage()` and `ListDevices()` functions serve as utility tools for users, and are executed in accordance with provided command-line arguments.

`PrintUsage()` displays supported command-line options and helpful information, thereby enhancing overall system usability. On the other hand, `ListDevices()` method<sup>8</sup> employs `pcpp::PfRingDeviceList::getInstance().getPfRingDevicesList()` in order to list all the currently available PF\_RING devices, aiding in system configuration.

If the usage of auxiliary functions is explicitly specified, program terminates upon their execution.

#### 4.3.3.4 Rule Representations

In the server component of the system, rules are represented in a dual manner. This bifurcation entails a separation between the database representation of rules and their representation across various other software modules.

The database's rule representation hinges on the utilization of a `.fbs` schema, which delineates the principal attributes of the object. This schema, provided in listing 4.3, is employed by the database framework to engender requisite binding code and `Rule` class, thereby facilitating interaction with user-defined objects.

■ **Code listing 4.3** Schema in `.fbs` format used in the database representation of rules.

```
table Rule {
  id: long;
  bz_length: byte;
  bz_start_bit: short;
  crc_length: byte;
  crc_start_bit: short;
  dest_ip: string;
  dest_port: short;
  duration: int;
  duration_type: string;
  mac_length: byte;
  mac_start_bit: short;
  mode: bool;
  new_value: ulong;
  pdu_id: short;
  serialized_polynomial: string;
  protocol: bool;
  serialized_secret_key: string;
  signal_length: short;
  signal_name: string;
  signal_start_bit: short;
```

<sup>8</sup>An acceptable alternative to the `ListDevices()` function call is the usage of `ifconfig` command in terminal.



```

    src_ip: string;
    src_port: short;
    status: bool;
}

```

In this architectural layout, fields such as the `polynomial` and `secret_key` are denoted as serialized entities, with their data type designated as `string`. This decision stems primarily from the constraints of the used database framework, which does not natively support direct storage of numerical arrays or vectors.

Conversely, within the scope of other modules, rule representation is manifested through the distinct `FrameRule` class. This class, whose public member variables closely mirror those depicted in the aforementioned schema, deviates in its handling of the polynomial and secret key fields. In `FrameRule`, these fields are represented as a `std::vector<uint8_t>`, enhancing usability.

Additionally, the `FrameRule` class incorporates auxiliary template functions for the serialization and deserialization of these fields, as well as for conversion to and from an instance of `Rule` class.

#### 4.3.3.5 Database

The `DataBase` class, an integral component of the server side of a system, is a sophisticated abstraction layer over the `ObjectBox` database framework. This class is designed to encapsulate the fundamental database operations, emphasizing ease of use and efficiency. It is critical to note that this class does not handle logical checks which are the responsibility of the user through the provided API (refer to image 4.14).



■ Figure 4.14 Specification of Database class.



**4.3.3.5.1 Overview.** The `DataBase` class is implemented as a singleton, ensuring that only a single instance of the database exists throughout the application. This pattern is achieved through the use of a static method, `Instance()`, which returns a reference to the static instance of `DataBase`. The copy constructor and assignment operator are explicitly deleted to prevent copying of the instance.

**4.3.3.5.2 Concurrency Control.** Concurrency control in the `DataBase` class acts as a proactive strategy to ensure smooth data handling. It utilizes a mutable `std::shared_mutex`, allowing multiple threads to read simultaneously, while write operations are exclusively locked. This design enhances performance in read-heavy scenarios while maintaining data integrity during writes.

**4.3.3.5.3 Database Operations.** The class provides a range of database operations, each encapsulated in a method. These operations include checking database emptiness (`Empty`), verifying the presence of a specific rule (`Contains`), adding (`Add`)<sup>9</sup>, updating (`Update`), retrieving (`Get`), and deleting (`Delete`) rules. Additionally, it supports clearing all rules (`Clear`) and setting the state of a specific rule (`Set`). These operations are safeguarded with `std::scoped_lock`, ensuring thread safety.

**4.3.3.5.4 Exception Handling.** Exception handling is a critical aspect of the `DataBase` class. Each database operation is enclosed within a try-catch block. In the event of an exception, the error is logged using `LOG_ERROR`, and the exception is rethrown. This approach ensures that all exceptions are properly logged and managed, maintaining robustness.

**4.3.3.5.5 Initialization and Destruction.** The constructor of `DataBase` is private, signifying its singleton nature. It initializes the database connection and prepares the queries. During destruction, it closes the database connection and, if defined, removes the database files. This lifecycle management is crucial for resource management and data integrity.

**4.3.3.5.6 Active Rule Management.** The class provides specialized methods for managing 'active' rules. These include counting active rules (`GetActiveRulesCount`), retrieving the first active rule (`GetFirstActiveRule`), and fetching all active rules (`GetAllActiveRules`). These methods utilize `activeRuleQuery`, a unique pointer to an `ObjectBox` query, tailored to filter active rules currently stored.

### 4.3.3.6 Packet Parser

The `Capture` class implements a functionality of a Packet parser abstract component; it is vital for real-time packet processing in the system, is designed for high performance and scalability. This class encapsulates the functionality to analyze and to start the manipulation of network packets as they are received.

**4.3.3.6.1 Overview.** `Capture` class contains only one static method; it is described in details in paragraph 4.3.3.6.2. Within the very class, the nested `CptrThrdArgs` structure is pivotal for packet processing, encapsulating all essential data required for this task. It comprises several key components, each serving a specific function:

- `strategies (std::vector<std::unique_ptr<stgy::IStrategy>>)`: A vector containing unique pointers, each pointing to a strategy object that adheres to the `IStrategy` interface (refer to section 4.3.3.8). These strategies are applied to packets during processing. The use of unique pointers ensures sole ownership and proper resource management of these strategies.

---

<sup>9</sup>IDs assigned to rules are unique and managed by the `ObjectBox` itself.

- `dstIface` (`pcpp::PfRingDevice*`): A pointer to a `PfRingDevice`, used to send out processed packets. It represents the destination network interface for these packets, which may be identical to or different from the source interface.
- `operCmd` (`utils::SharedResource<bool>`): A thread-safe utility for sharing a boolean value among multiple threads. It signifies an operational command (START or STOP) and is crucial for controlling the packet processing flow in a multi-threaded environment. When it is set to `false`, the program behaves as a transparent gateway (STOP). In contrast, having the `operCmd` set to `true`, enables the direct traffic interference with forcing the program to act according to the active rules (START).

The very structure is shared among all the threads participating in the packet capturing & parsing, being created and initialized as a member variable of `SystemLogic` class (refer to section 4.3.3.3) and populated with strategies by `BaseServer` class, denoted in section 4.3.3.3. This design facilitates effective management of packet processing parameters.

**4.3.3.6.2 PacketArrived Method.** The `PacketArrived(...)` static method is central to the class's functionality, addressing the high-level packet processing problematics. It is invoked on multiple threads depending on their availability<sup>10</sup> for each batch of incoming packets (depends on the network utilization) and employs protocol-based filtering, layered parsing, and selective strategy application, having the packet(s) represented as pointers to a stream of 'raw' `uint8_t` data. This approach is deemed to be the most effective, since it avoids any copying of data, reducing the overhead.

Precisely, the overall process of packet parsing is provided in pseudo code in Algorithm 3.

---

**Algorithm 3** Packet Processing.

---

```

1: procedure PACKETARRIVED(packets, numOfPackets, args)
2:   if not args.operCmd.Read() then
3:     args.dstIface.sendPackets(packets, numOfPackets)
4:     return
5:   end if
6:   for i = 0 to numOfPackets - 1 do
7:     packet ← parse(packets[i])
8:     if not validProtocol(packet) then
9:       continue
10:    end if
11:    actStgyPresent ← false
12:    for each strategy in args.strategies do
13:      if strategy.IsActive() then
14:        strategy.Execute(packet)
15:        actStgyPresent ← true
16:      end if
17:    end for
18:    if not actStgyPresent then
19:      args.operCmd.Write(false)
20:      break
21:    end if
22:  end for
23:  args.dstIface.sendPackets(packets, numOfPackets)
24: end procedure

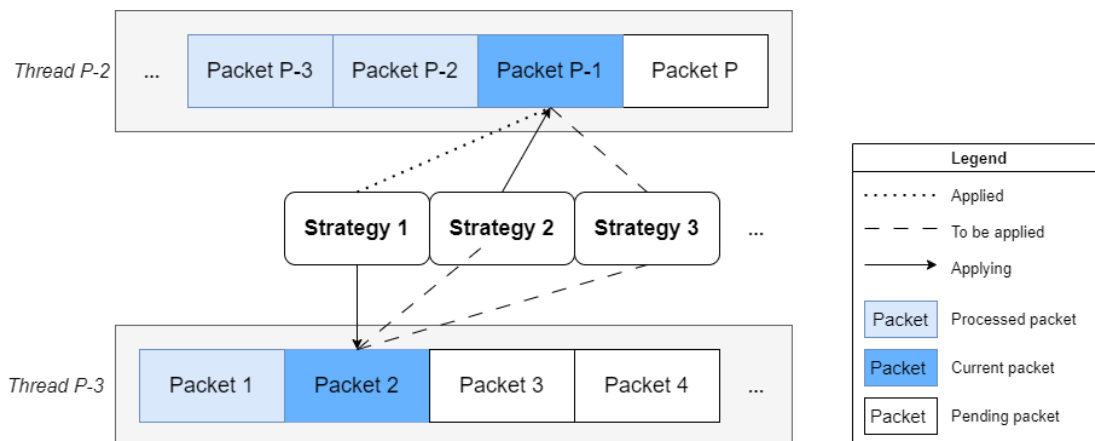
```

---

<sup>10</sup>The exact management of 'capture' threads is implicitly performed by PcapPlusPlus library.

The algorithm is quite intuitive and, summarizing, can be logically divided into the following phases:

1. **Operational Command Check:** Determines if processing should proceed based on the `operCmd` flag, allowing for immediate resending of packets in a STOP scenario (transparent gateway) or acting in compliance with the START (packet filtering / modification).
2. **Packet Processing:** Employs protocol checks (Ethernet, VLAN, IPv6) to filter packets, skipping non-conforming ones.
3. **Strategy Application:** Iterates over active strategies in `CptrThrdArgs` for applicable packet manipulation (refer to image 4.15).



■ **Figure 4.15** Application of strategies in multithread environment.

► **Note 4.10.** It is important to mention, that since `PacketArrived` method is being invoked on multiple threads, the actual strategy applications are being held concurrently as well. This implies that each strategy inside the `CptrThrdArgs` structure is being *shared* among all packet-processing threads.

4. **Active Strategy Presence Check:** Monitors active strategies; absence triggers a switch to STOP scenario by altering the operational command.

The `PacketArrived` method has a time complexity of  $O(NS)$ , where  $N$  is the number of packets and  $S$  is the average number of strategies. This complexity arises from the method's iteration over each packet and subsequent application of each active strategy. The space complexity is  $O(S)$ , dependent on the number of strategies.

This complexity profile ensures minimal processing overhead while having one strategy applied, making the system suitable for high-throughput networking environments. However, the application of several strategies simultaneously could, to some extent, raise an obstacle in terms of ASIL FTTI due to packet(s) processing overhead.

#### 4.3.3.7 JSON RPC Server

The `BaseServer` class, inheriting from `RpcServer`<sup>11</sup>, serves as the cornerstone of the system, initiating database operations with regard to their sanity and triggering the switch of AEPRIL-server operational mode. Key methods comply with an established client-server interface and include `UPSERT(...)`, `SET(...)`, `GET(...)`, `DELETE(...)`, `START(...)`, `STOP(...)`, and `TEST(...)`.

<sup>11</sup>`RpcServer` class is generated automatically by `Libjson-rpc-cpp` library according to the client-server interface denoted in `rpc-spec.json`.

First four methods execute database operations with prepared data, while the rest are dedicated to provide the means of system control. Exception handling is meticulously integrated into each method, ensuring robustness and stability: if an error or exception occurs, the proper message is sent to client. Additionally, the `ConstructFrameRuleObj(...)` method, a private utility function, servers for systematic construction of a `Json::Value` object (used for server replies) from a `FrameRule` instance.

Furthermore, `BaseServer` class stores a pointer to `CptrThrdArgs` structure (refer to paragraph 4.3.3.6.1) as a sole member variable, which is initialized at the constructor. This pointer is used at both of `START(...)` and `STOP(...)` methods to switch the operational mode and to either empty or populate the vector of active strategies.

### 4.3.3.8 Strategy Namespace

The namespace `stgy` is, without hyperbole, the most critical component of the server side of the system, tasked with the direct intervention in traffic through the execution of low-level operations on raw data. The fulfillment of functional requirements FR1-FR6 is intrinsically linked to the functionalities encapsulated within this namespace. Figure 4.16 exhibits the pivotal enumerations and data structures represented therein.

**4.3.3.8.1 Strategy.** In the design of the packet processing framework within the `stgy` namespace, the Strategy design pattern is deftly implemented to facilitate dynamic behavior based on predefined rules. Also, this design ensures the separation of created rules, easing the traceability of their application.

The abstract base class `IStrategy` serves as the cornerstone of this design, encapsulating common functionalities and providing an interface for concrete strategy classes, such as `ModifyStgy` and `FilterStgy`. The constructor of `IStrategy` initializes the strategy with a given `FrameRule`, setting up the foundational parameters and conditionally launching a detached thread for automatic deactivation based on time, depending on the rule properties. The rule itself is being stored to a member variable `mRule` at the time of strategy creation.

In this way, `IStrategy` can be described as a wrapper which manages the duration of the rule application, as well as its application nuances, without changing the rule itself.

The `Execute(pcpp::Packet &packet)` method, declared as a pure virtual function, is overridden in the derived classes to implement specific packet processing behaviors. `ModifyStgy` incorporates signal modification logic, while, in contrast, `FilterStgy` is dedicated to packet filtering tasks. This polymorphic design illustrates the application of the 'Open/Closed Principle'[68], enabling the extension of system capabilities without modifying the existing code base.

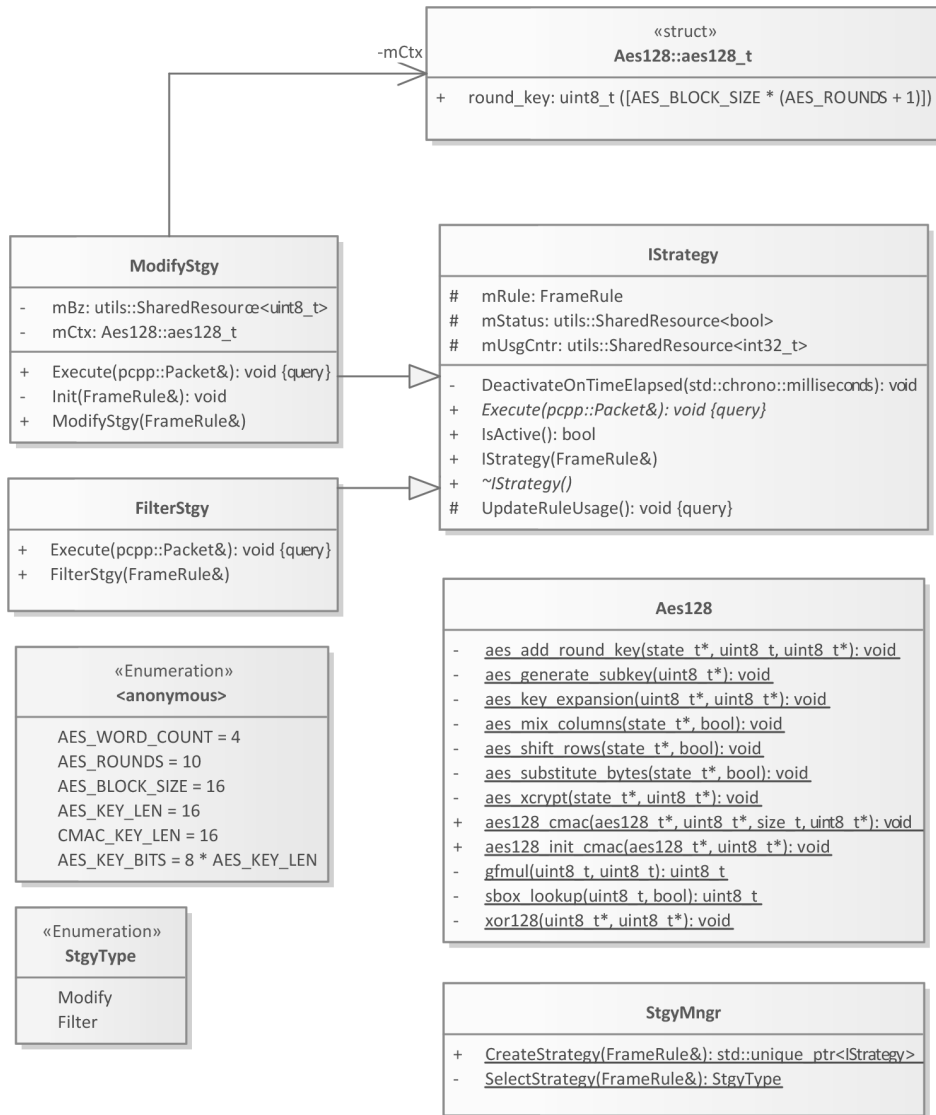
Significantly, the `IStrategy` class employs `utils::SharedResource` for managing shared data such as usage counter (`mUsgCntr`) and active status (`mStatus`), ensuring thread-safe operations essential in a concurrent environment. Member variable `mStatus` is equal to the rule status by default and can be changed during the program lifecycle if the 'duration\_type' of an active rule is not 'inf'. The method `UpdateRuleUsage` in `IStrategy` is pivotal for tracking the usage of strategies via `mUsgCntr`, particularly in cases, when the cyclical rule application is explicitly selected ('duration\_type' set to 'cyc'), highlighting the system's capability to handle real-time data processing requirements efficiently.

The architecture thus demonstrates a blend of modern C++ practices, including smart pointers, multithreading, and polymorphism, aligned with the demands of high-performance network applications. The adopted design pattern allows for systematic expansion and flexible adaptation of packet processing features.

**4.3.3.8.2 Strategy Manager.** The Strategy Manager (`stgy::StgyMgr`) class, is defined within the `stgy` namespace, and serves as a factory for creating strategy objects, adhering to the principles of object-oriented design. The factory method, `CreateStrategy(const FrameRule`

`&rule`), takes a constant reference to `FrameRule` as an input parameter, determining the strategy type required for execution.

The `StgyType` enumeration within `StgyMgr` delineates the available strategies: `Modify` and `Filter`. The decision logic, encapsulated in the private static method `SelectStrategy`, utilizes the state of the `FrameRule` object to ascertain the appropriate strategy type. Specifically, this method evaluates the `mode` attribute of `FrameRule` to decide between signal modification (`StgyType::Modify`) and traffic filtering (`StgyType::Filter`). This bifurcation of functionality ensures possible extensibility and maintainability of the system.



■ **Figure 4.16** Featured compounds of stgy namespace.

Subsequently, `CreateStrategy(...)` instantiates the strategy object with employment of `std::make_unique(...)` method, which returns a `std::unique_ptr<IStrategy>`. This use of smart pointers underscores modern C++ memory management practices, safeguarding against memory leaks and dangling pointers, thereby enhancing the robustness of the implementation. The return of a unique pointer to an `IStrategy` interface ensures that the client code remains

decoupled from concrete strategy implementations, aligning with the 'Dependency Inversion Principle'[69] and promoting a high level of modularity within the system architecture.

**4.3.3.8.3 Modification Strategy.** The `ModifyStgy` class implements the `IStrategy` interface within the `stgy` namespace; it is specifically designed for signal modification in packet processing. The constructor initializes the strategy by invoking the `Init(...)` method with a `FrameRule` object, making (`SharedResource<uint8_t> mBz`) member variable equal to 0 and setting up the AES128 cryptographic context (refer to paragraph 4.3.3.8.5) if a 'Secret Key' is provided.

The core functionality is encapsulated in the `Execute(...)` method, which serves for processing of `pcpp::Packet` object; the pseudo-code of this method is provided in Algorithm 4.

---

**Algorithm 4** Packet Modification.

---

```

1: procedure MODIFYPACKET
2:   payloadLayer ← PREMODIFICATIONCHECKS(packet, rule)      ▷ Is packet supported?
3:   if payloadLayer is valid then
4:     payload ← payloadLayer.GETDATA
5:     payloadLength ← payloadLayer.GETDATALEN
6:     pduFound, pduPosition, pduLength ← GETPDUINFO(payload, payloadLength, rule)
7:     if pduFound then
8:       WRITEDATA(signal modification parameters)
9:       if BZ is used then
10:        bz ← HANDLEBZVALUE(bz parameters)
11:        #ifdef BZ_AUTO_INCREMENTATION
12:        AUTOINCREMENTBz(bz, rule, payload, pduPosition)
13:        #endif
14:      end if
15:      if CRC is used then
16:        crc ← CALCULATECRC(crc parameters)
17:        WRITEDATA(crc, crc parameters)
18:      end if
19:      if MAC is used then
20:        mac ← CALCULATEMAC(mac parameters)
21:        WRITEDATA(mac, mac parameters)
22:      end if
23:      UPDATERULEUSAGE
24:      packet.COMPUTE/CALCULATEFIELDS      ▷ Recalculate packet checksums
25:    end if
26:  end if
27: end procedure

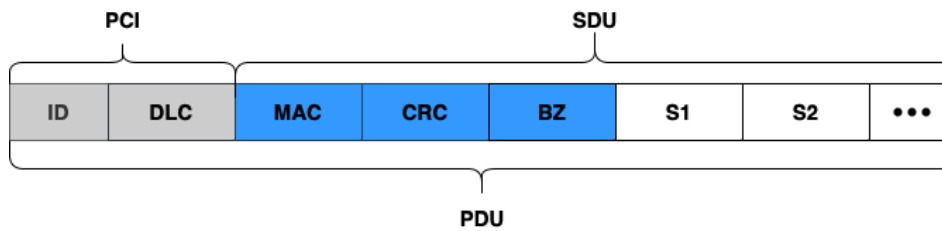
```

---

► Note 4.11. The circumvention of the in-PDU Alive Counter, as pertains to the functional requirement FR2, implies that any PDU falling under a specific rule should not be deemed outdated due to its BZ being less than that of a previous similar PDU. Under normal conditions (where packets are transmitted from sender to receiver without delay), such a scenario is unlikely. However, this can occur during packet processing under conditions of high network load. Specifically, if a capture thread processing multiple packets received simultaneously at time point  $T$  takes longer than another capture thread processing a single packet received at  $T + 1$  (in the current implementation all the packets received by a capture thread are being resent *simultaneously*).

The `Execute(...)` method conducts a series of operations on the packet data. Initially, it performs pre-modification checks to ascertain the applicability of the rule to packet which is

currently processed. If the packet's properties (protocol, source IP address, source port, etc.) do not match the rule, the packet is left as it is.



■ **Figure 4.17** Assumed L-PDU structure, the program is intended to work with, where each of blue-marked security mechanisms is optional.

Subsequently, the method searches for the desired PDU (refer to Figure 4.17) within the packet. In case of failure, packet is not modified. On the other hand, in case of success, signal modification is then executed by updating the selected signal with a new value.

Later, the presence of in-PDU security mechanisms is presumed according to the rule and their circumvention is performed depending on it:

- **BZ:** *BZ auto-incrementation* is intended to mitigate scenario described in Note 4.11. Initially, a local variable `bz` is set to 0. If the currently active rule dictates the presence of a BZ counter inside the PDU, its value is read into `bz` variable. From now on, two options are possible: either to perform an automatic incrementation of `mBz` variable for this strategy within the system, or not.
  - ▶ Note 4.12. The exact choice is being made at the time of program compilation, depending on macros defined in `conf.h` file, therefore making it impossible to change the behavior during run-time.

As was stated in Note 4.10, a comprehensive auto incrementation of `mBz` (*lastly used* in-PDU alive counter) must be held in concurrent environment, and that is the reason why the very variable is declared as `SharedResource`. In practice, plausible maximal values of in-PDU alive counter do not exceed 255 (or even 16 for some PDUs), making it possible to fit the entire value into one byte<sup>12</sup> (a variable of `uint8_t` type). This, in turn, means that auto incrementation of BZ counter requires the implementation of circular rotation of values within a fixed numerical range, and consequently, arises the necessity of usage of modular arithmetic as following:

1. The current value of `bz` is obtained by `mBz.Read()`, denoted as  $b_{\text{current}}$ .
2. This value is incremented by one, moving to the next sequential state:  $b_{\text{next}} = b_{\text{current}} + 1$ .
3. The modulus base is computed as  $2^n$ , where  $n$  is the bit length of `bz`, obtained from `mRule.bz_length`.
4. The final value of `bz` is then determined by applying the modulo operation:

$$\text{bz} = b_{\text{next}} \bmod 2^n$$

This ensures that `bz` cyclically wraps within the range  $[0, 2^n - 1]$ .

<sup>12</sup>In the current implementation, scenarios in which the in-PDU alive-counter values may exceed the value of 255, thereby surpassing the one-byte limit, are not encompassed. This limitation signifies that the system does not support the processing of alive-counter values that would require a representation beyond the capacity of an 8-bit data structure.



After the new value for the BZ counter was computed, it is stored back to `mBz` variable and written into the packet payload instead of the previous value. The denoted approach ensures, that each packet-processing thread always possesses the most recent used value of BZ counter within the same PDUs, transmitted over network. Additionally, the BZ auto incrementation is identified as the only feature bringing a relatively small processing overhead into the simultaneous applications of strategies in concurrent environment.

- **CRC:** Recalculation of in-PDU CRC partially depends on the read or computed (if auto-incrementation of `mBz` is enabled) BZ counter value and does not depend on the previous CRC (for details refer to paragraph 4.3.3.8.5). Therefore, the memorizing of actual (obsolete, after signal modification) in-PDU CRC value is not required. Additionally, this implies that the very CRC calculation differs from PDU to PDU (different BZ counters), providing additional means for security of in-vehicle communication. If the usage of in-PDU alive mechanism is not applied to a currently processed PDU, then initial value of `bz` variable (0) is used for CRC calculation, resulting into the shared approach for CRC calculation among all identical PDUs.

► Note 4.13. In such a way, it is noteworthy to mention, that the potential user of the developed software is supposed to be aware of placing the desired coefficient used for CRC calculation at the foremost place in the polynomial array, however, ensuring it maintains the predefined size of sixteen elements.

After the computation, the newly calculated CRC value is overwritten in the same location within the packet's payload, replacing the old CRC value.

- **MAC:** Recalculation of MAC does not depend on any side values apart from provided 'Secret Key'; this marks the extraction of previous MAC value from PDU payload as useless. The very MAC recomputation is performed by `AES128` class, also belonging to `stgy` namespace (refer to paragraph 4.3.3.8.5). After the new MAC was calculated, it is written to the position of the obsolete one.

The strategy concludes by invoking `UpdateRuleUsage()` to update usage metrics and executing `packet.computeCalculateFields()`, provided by `PcapPlusPlus`, for recalculating frame check sequences and other checksums.

**4.3.3.8.4 Filtering Strategy.** The `FilterStgy` class, inheriting from the `IStrategy` interface, is pivotal in the packet filtering mechanism. The realization of this class is directly connected to fulfilling functional requirement FR6. This is due to the fact that the filtering mechanisms available in the `PcapPlusPlus` library do not encompass the capability to check for the existence of a specific PDU within a packet.

The primary responsibility of the very class is to execute the filtering strategy based on predefined rules encapsulated within the `FrameRule` object. The constructor of `FilterStgy` takes a `FrameRule` object and initializes its base class, `IStrategy`, with this rule.

The core functionality of this class is encapsulated in the `Execute(...)` method. This method takes a reference to a `pcpp::Packet` object and performs conditional checks and operations on it. Initially, a `PreModificationChecks(...)` function is called, determining whether the packet meets criteria defined in `mRule` (refer to paragraph 4.3.3.8.5). If the packet satisfies these conditions, the method proceeds to check if a specific PDU ID is present (`mRule.pdu_id`).

When the PDU ID is set within the `mRule`, the method retrieves the payload and its length from the packet's payload layer. It then invokes `GetPduInfo(...)`, which attempts to locate the specified PDU within the payload, checking if the rule is matched.

Independently of the exact rule specification, if packet is matched, the method *clears* the raw packet data and updates the rule usage statistics through `UpdateRuleUsage()`.



► Note 4.14. The raw packet data are cleared using `.getRawPacket()->clear()` construct, essential to objects of `pcpp::Packet` type. It is worth mentioning, that formally packet is not being excluded from traffic at this stage; instead, the packet payload is being 'freed' and nullified causing the packet to be *empty*. Later, the driver of a network interface simply omits the sending of an empty packet, implicitly excluding it from traffic. The selected approach seems to be the easiest among the possible ones; additionally, it was practically validated during the very development of AEPRIL.

This design allows for a flexible and efficient packet filtering strategy, adaptable to various and conditions. `FilterStgy` encapsulates its logic cleanly, adhering to the principles of modularity and single responsibility.

**4.3.3.8.5 Strategy Elements.** Within the namespace `stgy`, a series of functions are intricately designed to address specific low-level challenges in processing of Automotive Ethernet packet. They serve as foundational elements that collectively formulate particular strategies, therefore being independent of any strategy instantiations. Those functions do not belong to a specific class, however, being a part of the `stgy` namespace, and therefore they are not denoted at Figure 4.16.

**4.3.3.8.5.1 Auxiliary Functions.** To begin with, the `PreModificationChecks(...)` function serves for packet verification, employing conditional checks on protocols, IPv6 addresses and UDP/TCP port numbers. Packet-level information is compared with the properties of a rule (if they are set), the strategy is operating with. The function returns a boolean value: `true` if the processed packet partially<sup>13</sup> matches the rule, `false` otherwise.

Additionally, functions `ReadData(...)` and `WriteData(...)`, are particularly notable in the code's bit-level manipulation capabilities. To be precise, `ReadData(...)` efficiently extracts specified bits from specified bitstream, while `WriteData(...)` accurately substitutes selected bits with provided value within it.

Furthermore, in contrast to `ReadData(...)` function, `GetBits(...)` serves as efficient tool for extracting selected bits from variables, rather than from bit streams; it is used to truncate MAC after it's computation in particular. Function's dual functionality in extracting either the most significant or least significant bits adds versatility to the data parsing process.

**4.3.3.8.5.2 PDU Examination.** The `GetPduInfo(...)` function is a high level abstraction, performing the nuanced handling of PDUs in respect to their header size. The handling of PDUs is adeptly managed with two distinct approaches conditioned by the `PDU_LONG_HEADERS` and `PDU_SHORT_HEADERS` directives defined in `conf.h` header file. The very function invokes either `ProcessLongHeaderPdus(...)` or `ProcessShortHeaderPdus(...)` upon return, depending on the definition of mentioned macros (conditional program compilation).

The `ProcessLongHeaderPdus(...)` function processes PDUs with extended header formats. It accepts parameters including a pointer to the packet payload, payload size, a `FrameRule` object (`rule`), and pointers to store the position and length of the matched PDU. The operation involves iterating over the packet payload, extracting a 32-bit PDU identifier and Data Length Code. Upon finding a match with the `rule`'s PDU identifier, the function updates `position` and `length` to reflect the PDU's location and size, then returns `true`. The iteration terminates upon a successful match or reaching the end of the payload. If the required PDU identifier was not found, the function returns `false`.

Contrastingly, `ProcessShortHeaderPdus(...)` addresses PDUs with shorter header formats, while the parameter structure and performed actions mirror `ProcessLongHeaderPdus(...)`. In this function, the PDU identifier is 24 bits and the Data Length Code is 8 bits, aligning with the shorter header specification.

---

<sup>13</sup>A function performs no checks for a presence of specific PDU inside the packet payload.

**4.3.3.8.5.3 CRC Calculation.** The `CalculateCrc(...)` function is designed to compute the Cyclic Redundancy Check (CRC) for a given Protocol Data Unit (PDU) in packet payloads. The algorithm<sup>14</sup>, reflecting the computation process, is provided as pseudo-code in Algorithm 5.

---

**Algorithm 5** In-PDU Cyclic Redundancy Check (CRC) Calculation.

---

```

1: procedure CALCULATECRC(input_bytes, input_length, scode, bz)
2:   crc  $\leftarrow$  0xFF
3:   cb_crc_poly  $\leftarrow$  0x2F
4:   for byte_index  $\leftarrow$  1 to input_length - 1 do
5:     crc  $\leftarrow$  crc  $\oplus$  input_bytes[byte_index]
6:     for bit_index  $\leftarrow$  0 to 7 do
7:       if crc & 0x80 then
8:         crc  $\leftarrow$  (0xFF &  $\neg$ (crc << 1))  $\oplus$  cb_crc_poly
9:       else
10:        crc  $\leftarrow$  0xFF &  $\neg$ (crc << 1)
11:      end if
12:    end for
13:  end for
14:  crc  $\leftarrow$  crc  $\oplus$  scode[bz]
15:  for bit_index  $\leftarrow$  0 to 7 do
16:    if crc & 0x80 then
17:      crc  $\leftarrow$  (0xFF & (crc << 1))  $\oplus$  cb_crc_poly
18:    else
19:      crc  $\leftarrow$  0xFF & (crc << 1)
20:    end if
21:  end for
22:  crc  $\leftarrow$  crc  $\oplus$  0xFF
23:  return crc
24: end procedure

```

---

The function operates on an array of bytes, representing the PDU, and applies a specific polynomial, defined as `cb_crc_poly`, to each byte. The CRC calculation begins by initializing the `crc` variable to `0xFF`. It then iterates over each byte of the input, excluding the first byte, and performs a bitwise exclusive OR (XOR) operation with the current CRC value. Following this, the function executes a nested loop for bit-level manipulation. If the most significant bit (MSB) of the current CRC is set, the function applies a left shift operation followed by an XOR operation with the polynomial. Otherwise, it simply performs the left shift. After processing all input bytes, the function incorporates a supplementary code from a given vector `scode` ('polynomial' denoted by rule) at the specified index `bz`. This additional step adds robustness to the CRC calculation. Finally, it performs another bitwise XOR operation with `0xFF` to complete the CRC computation, returning an 8-bit unsigned integer representing the CRC value.

The time complexity of the `CalculateCrc(...)` function is primarily determined by its nested loops. The outer loop iterates over each byte of the input array, excluding the first byte, which results in  $O(n - 1)$  iterations, where  $n$  is the length of the input array. Within this loop, there is a nested loop that iterates exactly 8 times for each byte, corresponding to the 8 bits in a byte, leading to a constant time operation of  $O(8)$  or simply  $O(1)$ . Thus, the total time complexity of the outer loop and its nested loop is  $O(n) \times O(1)$ , which simplifies to  $O(n)$ .

---

<sup>14</sup>The provided algorithm is *representative*, being provided for illustrating purposes only, and differs from those, employed by Porsche Engineering Services s.r.o. The exact algorithm used in commercial cars developed by Volkswagen (VW) Group companies resembles the provided one (in terms of performed actions and complexity), being concealed due to security & data protection considerations.

Additionally, there are constant time operations outside the loops, such as initialization and bitwise operations, which do not significantly affect the overall complexity. Therefore, the overall time complexity of the `CalculateCrc(...)` function is linear, denoted as  $O(n)$ , making it efficient for processing large PDUs.

**4.3.3.8.5.4 AES128-CMAC Computation.** The `Aes128` class, internal to `stgy` namespace (refer to Figure 4.16), represents a *confined* implementation of the AES-128 encryption & AES-based CMAC generation algorithms, in accordance with the specifications in FIPS PUB 197 [70]. The algorithms in question, given their status as an established and widely accessible entity with detailed specifications in the public domain, do not represent an innovation conceived within the scope of this work. Consequently, an in-depth focus on the principles of their operation is not deemed necessary; the featured aspects of implementation are detailed instead.

The class has no member variables and includes only `static` functions to ensure the possibility of their usage without having a class instantiated. `Aes128` is structured with various constants defined under an enumeration, including `AES_WORD_COUNT`, `AES_ROUNDS`, `AES_BLOCK_SIZE`, and `AES_KEY_LEN`, which align with the standard AES parameters. The `state_t` union, representing the AES state, is a 4x4 array, integral to the AES processing steps.

Key expansion, critical to AES, is implemented in the `aes_key_expansion(...)` function, following the algorithmic steps detailed in [71]. This method populates the round key array, essential for each encryption round, using the original cipher key ('Secret Key') and the AES S-Box for byte substitution, accessed via the `sbox_lookup(...)` function. This function showcases a pre-calculated `static constexpr` S-Box (RS-Box) approach, facilitating efficient byte substitution, a vital non-linear transformation in AES [72].

Byte substitution using S-Box (inverse S-Box) is handled by `aes_substitute_bytes(...)` function, introducing non-linearity into the system [73]. The methods `aes_shift_rows(...)` and `aes_mix_columns(...)`, responsible for permuting rows and mixing column data respectively, are pivotal for ensuring the diffusion property of AES [74]. In particular, the `aes_mix_columns(...)` function employs the `gfmul(...)` method for *Galois Field* (GF) multiplication, crucial for mixing data within columns as per AES specifications [75]. The XOR operation, a fundamental part of the AES round process, is implemented in `aes_add_round_key(...)`, effectively combining the state with round-specific keys. The comprehensive encryption process is encapsulated in `aes_xcrypt(...)` function, sequentially applying these transformations and complying with the AES encryption paradigm [70].

► Note 4.15. In accordance with the criteria delineated in the specifications [76], the exclusion of AES-128 based decryption from the implementation is a deliberate choice. This approach is justified by the fact that the primary objective of the developed class, namely the generation of MAC, does not necessitate the incorporation of AES-128 decryption processes.

In summation, as previously articulated, the `Aes128` class has been augmented to encompass the generation of AES-based CMAC, adhering to the RFC 4493 standards [76]. The initial phase in this process is 'context initialization', which is accomplished through the invocation of the `aes_init_cmac(...)` function. This function requires a 'Secret Key' (provided by the rule a strategy is operating with) and is executed concomitantly with the instantiation of a specific `ModifyStgy` instance and, therefore, prior to application of a strategy itself.

► Note 4.16. The AES context is strategically encapsulated as a member variable within a `ModifyStgy` class, rather than being a constituent of the `Aes128` class itself. This architectural decision ensures that each thread executing the respective strategy can concurrently and securely generate a MAC, utilizing the capabilities of the class, without necessitating any modifications to member or local variables after the context has been set up.

Finally, the generation of the message authentication code is accomplished through the execution of the `aes128_cmac(...)` function, initiated by an instance of `ModifyStgy`. This method involves generating subkeys using `aes_generate_subkey(...)`, aligning with the CMAC

specification for key derivation. The subsequent XOR operations and AES encryptions in `aes128_cmac(...)` culminate in the generation of a message authentication code.

## 4.4 Conclusion

In summary, this section of the thesis presents a comprehensive description of the developed software, serving as a technical guide that bridges the theoretical aspects and the internal workings of the system.

Throughout the software development process, a deliberate and well-reasoned choice was made to adopt a client-server architecture. Subsequently, the technologies employed in its creation were thoroughly examined and elucidated.


The initial step in the development of the final system involved defining the client-server interface, which elucidated the interaction between the discrete components of the system and enabled their independent development. Following this, the internal architecture of both the client and the server were individually delineated, explicated, and implemented.

Each noteworthy aspect of the implementation was thoroughly explained, and the some inherent disadvantages identified. All the details and challenges encountered during the development process were meticulously documented, and the solutions employed were logically justified. Additionally, a complexity analysis of some of the algorithms devised was conducted. It has shown, that the complexity inherent to server-side packet parsing process (refer to paragraph 4.3.3.6.2) could possibly represent the main drawback of the implementation, known at this stage of research.

Furthermore, the proposed implementation of the server-side component exhibits additional limitations in its scope. Specifically, in scenarios where multiple signals within a single PDU, safeguarded by security measures such as CRC, MAC, or BZ, require modification, the security algorithms would necessitate multiple recalculations. Ideally, a single recalculation post all signal modifications would suffice, being coherent with signal groups rather than individual signals, demanding the implementation of more sophisticated rule application mechanism. While the implemented approach may not be deemed efficient, it is considered adequate for *assessing the feasibility* of successful signal manipulations.

Notably, the final implementation of the ARXML Parser abstract component (inherent to client-side of the system), somewhat exceeds the scope initially set by the thesis's requirements. This is due to a deliberate adoption of a more generalized approach to the problem of data extraction from ARXML files, suggesting the potential reusability of this component in further scenarios. This aspect can be seen as a positive feature of the proposed solution, additionally aligning with modular design principles.

In conclusion, it can be asserted that the developed software is supposed to fully meet the stated functional requirements, by having all the required features implemented. The fulfillment of non-functional requirements cannot be determined at this phase, necessitating the evaluation of performance of the software at the time of its operation.



## Chapter 5

# Testing

This chapter aims at verifying if the developed software meets the initial requirements, and validating its functionality. To achieve the intended objective, a theoretical background for testing is reviewed and a corresponding testing strategy is being designed accordingly. Later, the software is being tested in compliance with it and the results are being presented and analyzed.

### 5.1 Theory of Testing

In a general sense, the testing of the software aims at both verifying the absence of software defects by validating its functionality in context of the requirements set. Commonly, three possible causes of software defects are differentiated [77]:

- **Error or mistake:** a human action that produces an incorrect result.
- **Defect, fault or bug:** a flaw in code, software, system or document that can cause it to fail to perform its required function (.g. incorrect statement or data definition).
- **Failure:** actual deviation of the component or system from its expected delivery, service or result.

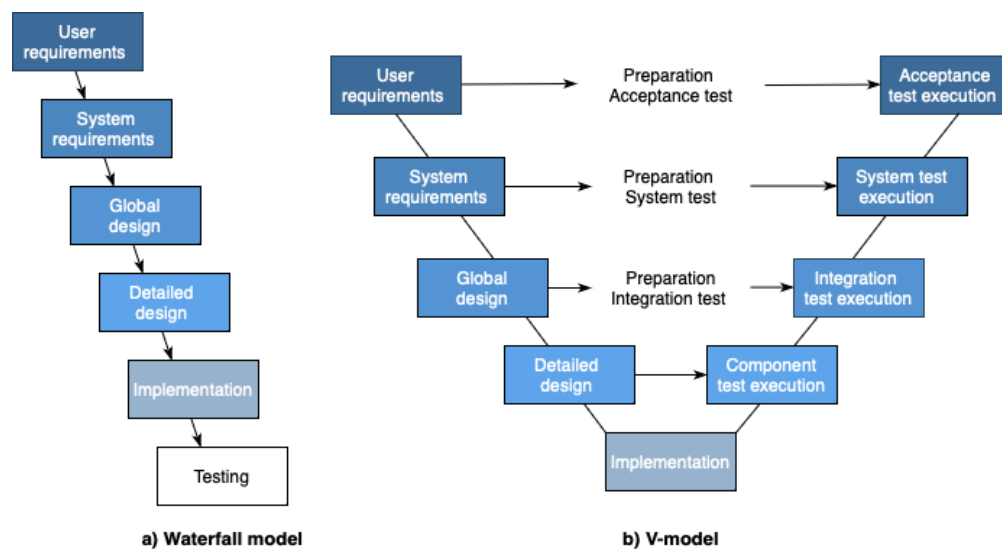
Several testing standards & software development models exist to guide the testing process in order to minimize the probability of software defects. For instance, standard ISO 9126 ('Software Engineering - Software Product Quality') contains the guidance on specifying, measuring and testing quality characteristics.

In addition, in contrast to traditional and well-known Waterfall Model of software lifecycle, the V-Model (widely used in automotive industry) addresses a significantly detailed testing approach (see Figure 5.1). Although variants of the V-Model exist, a common types of V-model uses four *test levels*, each with their own objectives [78]:

- **Component testing** searches for defects in, and verifies the functioning of, software units (objects, classes, etc.) that are separately testable. Component tests are typically based on the requirements and detailed design specifications applicable to the component under test, as well as the code itself. Usually they uncover defects such as incorrect code, logic & functionality and data flow problems.
- **Integration Testing** tests interfaces between individual components and interactions of different parts of the system(s). Integration tests are typically based on the software and system design (both high-level and low-level), the system architecture (especially the relationships between components or objects) and the workflows or the use cases. Consequently, such

tests reveal software defects coherent with failures in communication between components: incorrect or missing data, interface mismatch etc.

- System Testing** is concerned with the behaviour of the whole system as defined by the scope of the development project. Testing of integrated system aims at verifying that it meets the specified requirements. At the scope of the the thesis, it may include tests based on functional or software requirements specifications, use cases or high-level descriptions of system behaviour, interactions with the operating system and system resources. The focus is on end-to-end tasks that the system should perform, including non-functional aspects, such as performance. The quality of the data may be of critical importance as well as the test environment since it should correspond to the final production environment as much as possible. Hence, the typical software defects revealed during performing system testing are: incorrect calculations, data and/or control flows, unexpected system functional or non-functional behaviour, different failures of system work in production environment and inability to carry out end-to-end functional tasks.
- Acceptance Testing** typically produce information to assess the system's readiness for release or deployment to end-users. It covers a number of activities such as validating of installation, maintenance, disaster recovery, checking for security vulnerabilities, performance and load testing, etc.



■ **Figure 5.1** Comparison of Waterfall (a) and V-Model (b) representations of a system's development lifecycle.

Broadly, software testing comprises both *static* and *dynamic* testing techniques. During the static testing, software is not being executed; rather the specifications, documentation and source code that comprise the software are examined in varying degrees of detail.

*Static analysis*, being descended from compiler technology, is a form of automated static testing that can check for violations of certain standards (predefined rules) and can find things that may or may not be defects. A good static analysis includes *data flow & control flow* analysis, statement and branch testing & coverage. It excels at identifying defects (variable reference with undefined value, unused variables, unreachable code, syntactic violations of code, etc.) that are difficult to notice using dynamic methods before they can affect the testing process.

Conversely, dynamic testing techniques usually include *use case testing*, *state transition testing*, *decision table testing*, *boundary value analysis* and *equivalence partitioning* [77].

Notably, the *test strategy* itself is defined as a high-level description of the test levels to be performed and the testing within those levels for an organization or programme [77].

A good testing strategy implies that each level is comprehensively tested using different approaches (test types) that, combined together, minimize the defect probability. A *test type* is a group of test activities based on specific objectives aimed at testing specific characteristics of a component of a system [77]. Depending on its objectives, testing is organized differently:

- **Functional testing** is the process of ensuring that a system or its components perform their intended functions correctly. This involves assessing the system's behavior to verify that it aligns with the expected outcomes.
- **Non-functional testing** goes beyond the core functionality of a system to evaluate various qualities. It encompasses different aspects such as performance testing to measure system speed and resource utilization, usability testing to guarantee an intuitive user interface, load testing to assess system behavior under varying workloads, security testing to identify vulnerabilities, and scalability testing to determine how well the system copes with increased loads and scaling challenges.
- **Structural testing** delves into the internal structure of the software. This involves methods like code coverage analysis to assess the extent to which the code is exercised by testing.
- **Regression testing** is a quality assurance process that focuses on ensuring that new updates or changes to the system do not adversely affect existing functionality. It involves retesting previously validated parts of the system to identify any unintended side effects or defects introduced by recent modifications.

In addition, **user interface testing** incorporates both *qualitative* and *quantitative* methodologies [79]. The quantitative approach is anchored in analyzing user interactions with the application and employs statistical methods.

In turn, qualitative testing bifurcates into two distinct methodologies: *user-involved* and *user-independent* testing. In scenarios where user-independent testing is preferred, an expert in the field typically conducts the assessment, employing a *heuristic analysis* method, predicated on compliance with established heuristic principles. Conversely, the user-involved qualitative testing is usually executed in a controlled laboratory environment [79].

In conclusion, all the aforementioned test types & levels, as well as depicted testing techniques and methodologies, are deemed to be necessarily applied in testing of the software designed for mass-market deployment and extensive user interaction. Conversely, considering the status of the developed software as a *functional prototype*, a testing strategy for it may not be comprehensive, focusing solely on specific aspects.

## 5.2 Formulation of Testing Strategy

Generally, the test strategy shall include an elaborated conjunction of *selected* test types performed at *featured* test levels. Since the developed system for signals manipulation on Automotive Ethernet is deployed utilizing a client-server architecture, the testing strategy implies the possibility of separated approaches for the functionality validation of both client and server, focusing on the most critical facets.

### 5.2.1 Client

To sufficiently test the server-side component of the system, it is considered essential to conduct direct assessments of the created graphical user interface, as well as the functionality verification of the ARXML Parser, as these specific parts of the resulting software have been identified as



warranting attention. In contrast, the development and implementation of component & integration tests for remaining components were deliberately excluded, predicated on the presumption that there are no software defects. Testing on the system level is suggested to be performed manually and simultaneously with the testing of the GUI.

Notably, the overall process of applying unit testing to ARXML Parser abstract component, involves the separate testing of several software units comprising it: *Query Handler*, *Query Builder*, *ARXML Data Extractor*, *Config Provider*, *Data Writer*, *Data Object*, *Data Value*, *Data Query*, *Parser* and *Tabularize function*.

Sometimes, the testing shall be done with non-empty `test.arxml` file, specially designed to verify the functionality of a component. Moreover, since the component was created while considering the possibilities of further deployment as a stand-alone tool, the testing shall include the verification of all declared functionalities. Additionally, branch & line coverage shall be analyzed, adhering to the structural testing principles

The non-functional testing is also an important aspect of evaluating the performance of ARXML Parser in real-world conditions. It shall be performed with focus on the speed of processing of *real* ARXML files (according to the schema template presented in paragraph 4.3.2.2.1.3) and shall mandatory evaluate the impact of `.arxml` size on its processing speed (since at the beginning it is the only known attribute of a file). The impact of number of signals & frames on the overall processing speed shall be evaluated as well, to analyze the resulting file processing time from perspective of its inner structure.

► Note 5.1. The speed of ARXML processing contextually include the data transformation issues and exporting of retrieved data into the targeted `.json` files, therefore covering the entire process of retrieval of signals specifications. The processing time is not anticipated to be small, and therefore the ultimate accuracy in measurements is not deemed necessary; it shall be measured with `time.time()` construct, inherent to `time` Python module.

In realm of user-independent GUI testing, the heuristic analysis shall be employed to analyze the qualitative characteristic of it. However, considering the absence of an expert in field, the analysis shall be conducted by the author of the thesis according to the the principles defined by Jacob Nielsen [79].

Furthermore, the user-involved testing in laboratory conditions shall be performed, adhering to the following test scenario:

1. **Selecting ARXML for processing**
2. **Checking the server connection**
3. **Inserting two new rules (both signal modification and traffic filtering)**
4. **Updating the duration of existing rule (any)**
5. **Changing the status of an updated rule**
6. **Obtaining the specification(s) of a selected rule and all uploaded**
7. **Starting the traffic interference**
8. **Stopping the traffic interference**
9. **Deleting all the uploaded rules**

The test shall be performed on two computers within a local network established, with client & server applications installed on them separately. The successful sending of the JSON-RPC request shall be verified manually with reviewing of the resulted log file of the server-side. Importantly, all the comments on the GUI & its utilization process are to be collected and analyzed.



## 5.2.2 Server

For the testing of the server-side component of the system, it is deemed necessary to conduct direct assessments of functional capabilities and non-functional qualities in real-world conditions, adhering to the system testing principles. Conversely, the creation & execution of component tests and integration tests has been intentionally omitted, based on the assumption of the absence of software defects (e.g. correct interface usage, dataflow).

Importantly, the primary objective of the testing is *to evaluate the feasibility* of successfully manipulating signals on Automotive Ethernet. For the execution of system testing, conducting the following test cases (in accordance with the program use-case scenarios outlined in section 3.2.1) is considered sufficient:

1. **Transparent Gateway**
2. **Signal modification**
3. **Signal modification + BZ auto incrementation**
4. **Signal modification + in-PDU CRC calculation**
5. **Signal modification + MAC calculation**
6. **Exclusion of specified packet from traffic**

The result of traffic interference (either it was successful or not), shall be verified using CANoe<sup>1</sup> of version 7.4.

During the testing process, emphasis is to be placed on assessing program performance under varying loads (both overall network utilization & number of active rules applied) and with different initial settings. This includes evaluating the performance depending on the number of threads used for parsing and forwarding internet packets, as well as the processing speed of the packets themselves. Furthermore, it's essential to appraise packet transmission speed contingent upon the number of active rules, to determine the scope of system usability in context of plausible supported ASIL FTTIs (refer to paragraph 2.1.2.1.1). In this context, the *20ms* boundary is to be treated as provided ASIL FTTI.

It is proposed to utilize various metrics for assessing the program's performance: packet processing time as well as network interface load (packets/s, kB/s). This approach will enable tracking not only the processing speed of a single packet, but also the impact of capture threads on their throughput, possibly revealing further observations. To mitigate measurement error, it is suggested to conduct measurements over a duration of 30 seconds, extracting the arithmetic mean to determine key values.

► Note 5.2. To simplify the task at hand, it is proposed to disregard the overhead (associated with hardware, drivers, operating system, the library in use, etc.) related to the reception and direct transmission of a packet. Instead, the focus is to be made on the execution time of the `PacketArrived(...)` method, which is to be measured as the difference between `std::chrono` timestamps.

► Note 5.3. The `sysstat` utility is to be utilized for the measurement of network interfaces average throughput, using the `sar -n DEV 1` command.

To facilitate this, the automation of the testing process is suggested, using a `bash` script. This script utilizes pre-prepared rules generated by the client-side of the system and engages in direct interaction with the server. In this context, direct server communication is proposed to be carried out using the `curl` utility.

---

<sup>1</sup>It is a development and testing software tool from Vector Informatik GmbH.

► Note 5.4. The interaction with the server is to be performed *iteratively* by sending of JSON-RPC requests in the following sequence: **DELETE** (all), **UPSERT** (variable number of times), **START** and **STOP**. This sequence implies that database of rules is being always erased before the actual run, the new rules are being uploaded and the traffic interference is being triggered. The number of rules to be utilized is proposed to increase *arithmetically* (starting from 1) until the certain bound, which is to be determined manually, depending on the program performance and the test case.

Finally, to ascertain the impact of the processor's qualitative characteristics on program performance, it is suggested to conduct tests on two different computers.

### 5.3 Functionality Verification & Validation

The testing activities performed, along with the results obtained, are detailed in the respective subsections according to the testing strategy outlined in the previous section. It is noteworthy to mention, that the defects (or problems) identified in the developed software were broadly classified depending on their severity into the following categories: *informational*, *low* (software remains usable), *moderate* (certain difficulties in using the software), and *high* (software is unusable or its use is limited).

Importantly, in order to perform testing of both the client and server server-side of the developed system, the following *available* laptop configurations (hereinafter, Computers) were employed, depending on the test type & level:

- HP ProBook 450 G8 (referred to as 'Computer A')
  - *Processor*: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz x8
  - *RAM capacity*: 16 GB
  - *Operating System*: Microsoft Windows 10 Pro x64 + Linux Mint 20 x64
- Lenovo IdeaPad Gaming 3 15IHU (referred to as 'Computer B')
  - *Processor*: 11th Gen Intel(R) Core(TM) i5-11320H @ 3.20GHz x8
  - *RAM capacity*: 16 GB
  - *Operating System*: Ubuntu 22.04.3 LTS x64 (Kernel version: 6.2.0-39-generic)

Both units have the same number of physical processor cores (four), with differences in operational frequencies. The laptops' graphics cards were not a factor in the experiments, as the processing of ARXML files and Ethernet packets, in context of the developed system, is conducted solely through the CPU, without any GPU involvement.

To ensure the ease of installation and consistent system setup (server-side in particular) across different machines, specific bash scripts were devised, also available in the thesis' attachments.

#### 5.3.1 Client

##### 5.3.1.1 Component & Structural Testing: ARXML Parser

All tests for the software units identified in the testing strategy were developed using the Pytest framework, version 7.4.4.

For each module under test, a suite of unit tests was composed, concentrating on the verification of various aspects of its functionality. This included assessing the module's behavior with both correct and incorrect input data, as well as testing identified boundary values for inputs

where applicable. The design of the tests emphasized modularity, targeting specific functionalities within the component under test. This modular approach facilitates the identification of potential issues and streamlines the maintenance and scalability of the test suite.

Additionally, several tests employed Pytest fixtures, which furnished a reusable set of data or states, enhancing the efficiency of the testing process. Execution of the compiled tests indicated that the developed program adeptly meets the established objectives, being prone to no known defects.

Furthermore, Coverage.py, version 7.4.0, was utilized for structural testing of selected modules, specifically to assess the extent of code coverage achieved by the unit tests. The outcomes of the testing process, along with the number of tests written for each module, are presented in Table 5.1.

Module (.PY)	Number of Test Cases	Statements	Missing	Branches	Partial	Coverage
parser	9	77	10	20	4	84%
config_provider	4	13	0	4	0	100%
data_writer	1	42	2	20	2	94%
tabularize	6	25	0	0	0	100%
data_object	5	31	3	11	1	86%
data_value	2	11	3	2	1	69%
data_query	9	43	14	14	3	63%
query_builder	11	99	3	39	2	96%
query_handler	18	34	6	10	2	82%

■ **Table 5.1** Summary on component and structural testing of ARXML Parser abstract component.

The tabulated results illustrate a significant disparity in software testing outcomes. While a considerable volume of tests, amounting to 55, has been written for certain modules, the overall code coverage achieved by these tests remains suboptimal. Nonetheless, given that the software is in a prototypical phase, the primary function of the tests can be interpreted as being *illustrative* rather than definitive. In this light, the tests serve as preliminary indicators of functionality and stability, with the results being deemed satisfactory for this stage of development.

### 5.3.1.2 Non-Functional Testing: ARXML Parser

In order to conduct non-functional testing of the ARXML parser, a collection of 120 authentic ARXML files, previously employed by automotive companies for delineating ECU communications within vehicular networks, was procured. It is important to note that prior to initiating the testing process, the author of this study possessed no knowledge regarding the purpose or contents of these files.

The testing itself was carried out on Computer A, equipped with a Windows operating system. The schema template, denoted in paragraph 4.3.2.2.1.3, was utilized during the *automated* testing process. The very results of it are presented in Table 5.2, as well as at Figures 5.4, 5.2 and 5.3.

The test results revealed that only a third of *available* files contained a pattern designated by the YAML schema, which described communication via Automotive Ethernet. Conversely, it was later discovered that the majority of the remaining files predominantly detailed communication aspects based on CAN and FlexRay technologies.

As illustrated in the provided table, it is apparent that files with larger sizes do not always require longer processing times (see obtained data for files 36, 37). This observation unequivocally suggests that factors other than file size, such as the number of signals and frames<sup>2</sup>, *possibly*

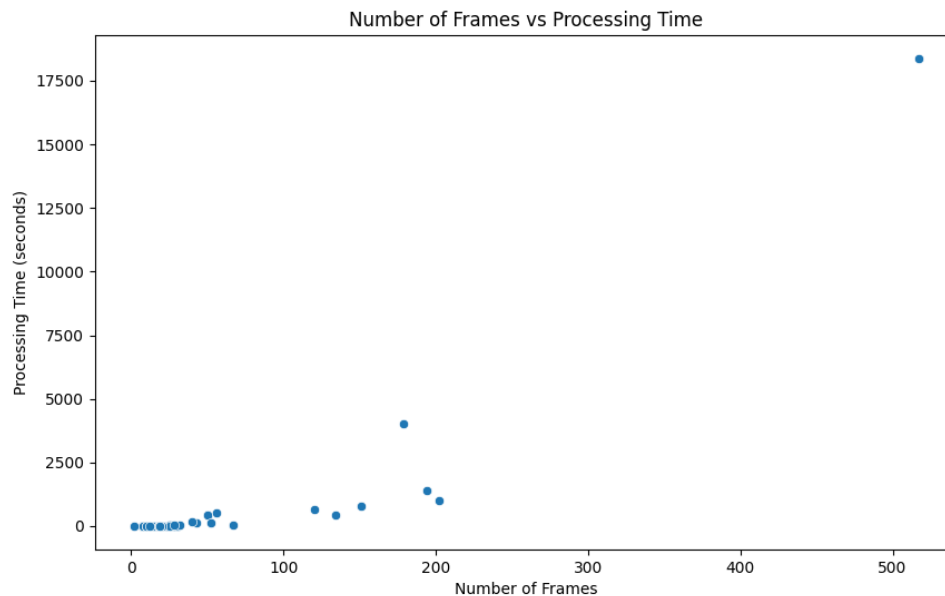
<sup>2</sup>Original frames, prior to data transformation handled in paragraph 4.3.2.2.4

*among others*, may play a significant role. To comprehensively unravel the relationships between file size, the number of frames, the number of signals within a file, and the time taken to process these files, a more detailed analysis shall be carried out.

File	Size (kB)	Number of Frames	Number of Signals	Processing time
1	2532	7	127	0 minutes 1 seconds
2	776	8	13	0 minutes 1 seconds
3	1270	2	83	0 minutes 1 seconds
4	1877	9	159	0 minutes 1 seconds
5	1880	9	159	0 minutes 1 seconds
6	13212	8	102	0 minutes 1 seconds
7	13214	8	102	0 minutes 1 seconds
8	10324	10	114	0 minutes 1 seconds
9	24003	10	114	0 minutes 1 seconds
10	8575	23	522	0 minutes 2 seconds
11	9968	18	1296	0 minutes 3 seconds
12	12740	28	848	0 minutes 4 seconds
13	9494	20	964	0 minutes 4 seconds
14	11013	19	1398	0 minutes 4 seconds
15	19933	12	319	0 minutes 4 seconds
16	11150	19	1317	0 minutes 4 seconds
17	14369	25	1088	0 minutes 6 seconds
18	18018	30	1259	0 minutes 8 seconds
19	24590	18	441	0 minutes 11 seconds
20	17216	18	2469	0 minutes 14 seconds
21	33335	25	1849	0 minutes 18 seconds
22	17726	24	1144	0 minutes 19 seconds
23	30642	15	3091	0 minutes 21 seconds
24	53424	32	2289	0 minutes 36 seconds
25	38314	25	2456	0 minutes 38 seconds
26	47744	28	2886	0 minutes 41 seconds
27	52048	67	1014	0 minutes 59 seconds
28	56289	52	2974	2 minutes 9 seconds
29	54820	43	4733	2 minutes 22 seconds
30	87059	40	5692	2 minutes 56 seconds
31	130324	50	6946	6 minutes 57 seconds
32	134528	50	7084	7 minutes 16 seconds
33	109126	134	4289	7 minutes 21 seconds
34	184279	56	5978	8 minutes 41 seconds
35	133848	120	5540	10 minutes 40 seconds
36	159155	151	8305	13 minutes 16 seconds
37	133402	202	10019	16 minutes 40 seconds
38	212416	194	7041	23 minutes 31 seconds
39	253635	179	11790	67 minutes 11 seconds
40	357007	517	41361	306 minutes 14 seconds

■ **Table 5.2** Results of non-functional testing of ARXML Parser, sorted by processing time in ascending order.

The analysis is conducted using statistical methods, particularly *Pearson correlation* and *regression analysis*, to ensure a rigorous and empirical evaluation of these relationships. Namely, Pearson correlation measures linear correlation between two sets of data, while regression analysis serves as a reliable method of identifying which variables have impact on a topic of interest.



■ **Figure 5.2** Dependency of ARXML processing time on the number of frames

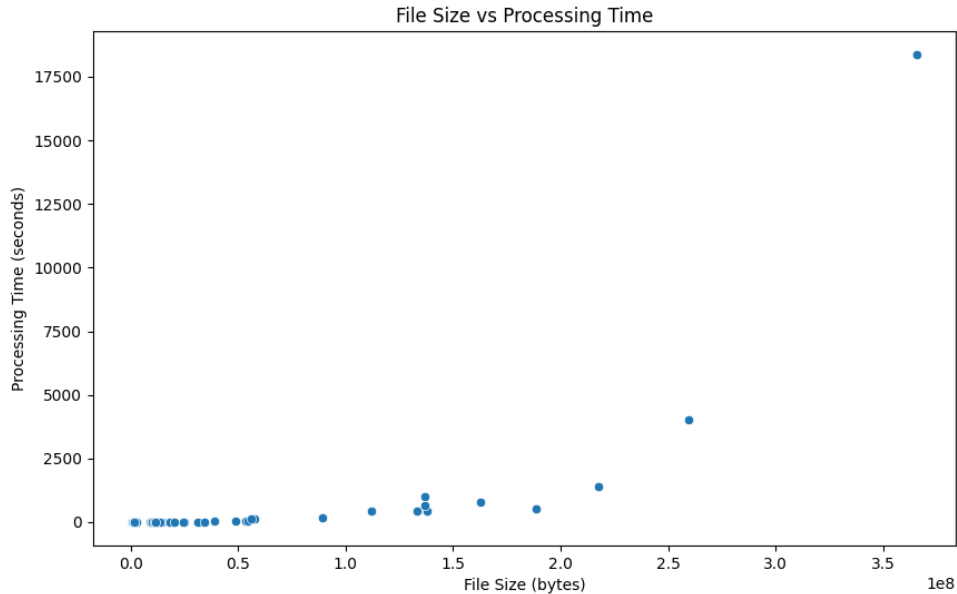
It is evident, that as the number of signals & frames increases, contributing to the total file size, so does and the resulting processing time. However, the key point of interest is to determine the factor among mentioned ones, having the biggest impact on the duration of ARXML file processing.

► **Note 5.5.** Given the limited size of the available file sample, which is insufficient for comprehensive representativeness in a robust statistical analysis, the results obtained hereafter should be regarded more as *indicative* rather than definitive.

The computed *linear* regression model, encompassing all three predictors, exhibits a notably high R-squared value of **0.9401**. This indicates that approximately 94% of the variability in the processing time is accounted for by the linear combination of three key variables: file size (in bytes), number of frames, and number of signals. This substantial proportion suggests a strong *linear dependence* of processing time on these factors collectively.

Analyzing each predictor individually, the separate linear regression models reveal varying degrees of explanatory power:

- The model with **file size** as a sole predictor has an **R-squared of 0.5325**, indicating a moderate explanatory capability, accounting for approximately 53% of the variability in processing time.
- The **number of frames**, as an independent predictor, results in an **R-squared of 0.7948**, suggesting a stronger linear relationship with processing time.
- The most pronounced individual predictor is the **number of signals**, with an **R-squared of 0.9041**, indicating that it alone can explain around 90% of the variability in processing time.



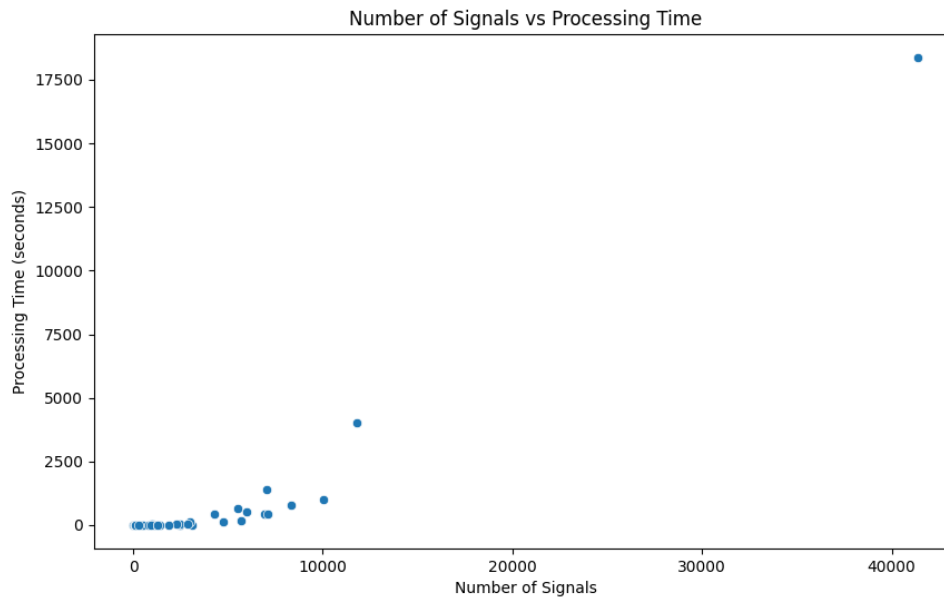
■ **Figure 5.3** Dependency of ARXML processing time on the file size.

Further, the Pearson correlation coefficients were calculated to measure the *linear relationship* between processing time and file size, number of signals & number of frames.

- The correlation coefficient between **file size** and **processing time** is **0.7298**. This indicates a *strong positive linear correlation*, suggesting that as the file size increases, there is a tendency for the processing time to increase as well. However, it is noteworthy that this correlation, while substantial, is not perfect, implying the influence of other factors on processing time.
- The correlation coefficient for **number of frames** versus **processing time** is **0.8915**, representing a *very strong positive correlation*. This implies a more pronounced linear relationship, where an increase in the number of frames in a file significantly impacts the processing time, more so than the file size alone.
- Furthermore, the correlation coefficient for **number of signals** versus **processing time** is **0.9509**, indicating an *extremely strong positive correlation*. This suggests that the number of signals in a file is a highly influential factor in determining the processing time, potentially more impactful than file size and number of frames.

Summarizing, it can be concluded, that the implemented ARXML Parser cannot be treated as effective (in terms of speed) in scenarios, where there is a need for processing of large AUTOSAR XML files. Among all the analyzed factors, the number of signals possesses the most significant impact on the overall processing time of ARXML file. Additionally, the in-ARXML structural elements not associated with the utilized schema template, also could bring further deceleration into the ARXML parsing, requiring extra consideration.

Despite the results are quite intuitive (ARXML Parser was tested in context of signal extraction capabilities), the obtained findings highlight that the potential and *relatively accurate* prediction of file processing time cannot be done based only on the disposable file size, requiring the knowledge of its inner organization.



■ **Figure 5.4** Dependency of ARXML processing time on the number of signals.

### 5.3.1.3 GUI Testing: Heuristic Analysis

Due to the absence of an expert in the field, the results of the conducted heuristic analysis may not necessarily be accurate and comprehensive, stemming from the limited experience. The findings are presented below:

1. **Visibility of System Status:** All actions that require a significant amount of time provide feedback to the user in form of a textual message. Active input fields and tabs are distinguishable, buttons respond immediately. The statuses of ongoing processes are visually indicated with a textual log, so that user is always aware of the current status. The resulting GUI is one paged, negating the needs for any visual transitions.
2. **Match Between the System and the Real World:** The user interface is primarily implemented in English language, with all labels accurately reflecting the meaning of the particular elements. No translation errors are present. All the information is presented in textual manner, designed to be intuitive and minimalist, corresponding the application's status of a technical utility. Only one icon is present at the window header.
3. **User Control and Freedom:** In context of user control, the detailed analysis of visual elements is available at section 4.3.2.1.1. Among the main disadvantages, the GUI lacks the possibilities of adjusting window size and manually erasing the output area. Graphical user interface does not comprise any confirmation components, neither provides means for altering the ongoing action (ARXML parsing, rule creation), therefore reducing the user freedom.
4. **Consistency and Standards:** Developed for desktop platforms (Windows, Linux, etc.), the GUI partially complies with Win32 standard<sup>3</sup>. While primarily adhering to Windows standard, some elements may not align with user interface standards on other platforms. Moreover, the exact layout of the user interface directly depends on the screen dimensions

<sup>3</sup>Fully available at [80].

due to implementation constraints. Some of the similar input fields are implemented using different approaches, causing inconsistencies in their on-screen appearance with varying screen resolutions. The GUI supports only a light theme, potentially leading to color discrepancies if the default system theme is dark. Standard colors and fonts are used, being easily distinguishable and readable.

5. **Error Prevention:** The user interface (as well as client-side in general) lacks actions that alter the state, thereby negating the need for confirmation actions in this context. Any logical errors and inconsistencies arising during the usage of a program are prompted to the output area, explaining the user the actual cause and, sometimes, the possible solutions.
6. **Recognition Rather Than Recall:** Users need not remember information, as after any performed action it is not automatically erased from the input fields neither from the output area. Since the output area is never erased, it provides historical feedback easing the general utilization of the software. None of the graphical elements can be hidden, possibly bringing intrusiveness.
7. **Flexibility and Efficiency of Use:** The program does not include keyboard shortcuts or macros, due to the lack of areas where they are applicable.
8. **Aesthetic and Minimalist Design:** The screen displays only relevant information. The user interface is designed with the minimal necessary number of functions and elements.
9. **Help Users Recognize, Diagnose, and Recover from Errors:** The simplicity of the graphical interface prevents unsafe actions, thereby obviating possible errors that require complex resolutions. Also, as it was noted, almost each error arising is supplemented with explanation message which is always visible at the output area.
10. **Help and Documentation:** No in-GUI assistance is available and some elements lack explanations or are not intuitive enough (e.g., ID), demanding the user experience and previous acquaintance with the software.

Additionally, the problems & defects identified during the heuristic analysis are summarized in Table 5.3.

Severity Lever	Defect
<i>Moderate</i>	Window size depends on the screen resolution; inability to change it.
<i>Low</i>	Consistency of UI for different platforms is not ensured.
<i>Moderate</i>	Inconsistency of element appearance on screen.
<i>Low</i>	Absence of help menu or documentation.
<i>Moderate</i>	Inability to stop the ongoing action.
<i>Low</i>	Inability to alter a color scheme.
<i>Low</i>	Inability to erase the output area.

■ **Table 5.3** User interface problems & defects identified by the heuristic analysis.

#### 5.3.1.4 GUI Testing: User-Involved Tests

The testing was conducted with six employees from the Vehicle Testing department of Porsche Engineering Services s.r.o. in a controlled laboratory setting. Participants were initially briefed on the objectives of the experiment.

Furthermore, half of the participants received a comprehensive explanation about each element of the graphical user interface to assess the influence of familiarity with the program documentation on their overall user experience. Such a decision is based on the previously conducted heuristic analysis.



Subsequently, each participant was individually tasked to execute a predetermined sequence of operations using a specifically prepared laptop. The test scenario and necessary input data were made available to participants via a text document pre-loaded onto the laptop (Computer A).

Post-experimentation, individual interviews were conducted immediately following the test sessions. These interviews aimed to gather participants' subjective experiences, comments, identified issues, and their perceived severity regarding the usability of the system. The responses obtained were methodically analyzed, and the identified issues were categorized according to their severity levels (refer to Table 5.4 for detailed classification). Overall, the obtained results partially comply with those, acquired with previously performed heuristic analysis.

Furthermore, the control group's observations revealed significant, but anticipated findings. Without prior explanation of the GUI elements, users encountered numerous difficulties, leading to incorrect usage instances. Thus, the intuitiveness of the GUI is brought into question, as the study indicates a need for integrated documentation to aid user interaction with the system, suggesting that the current GUI design is not entirely self-explanatory.

Additionally, several semi-critical bugs<sup>4</sup> were identified during the testing phase, primarily stemming from implementation flaws. A notable issue was the random unresponsiveness of the GUI during JSON-RPC request creation and dispatch. A proposed solution to this problem is the implementation of multi-threading, similarly to the process of `Worker` class utilization in ARXML parsing. This approach is proven to enhance responsiveness and efficiency in handling asynchronous tasks.

While the GUI is fully functional, there is a clear necessity for further refinement.

Severity Lever	Defect
<i>Low</i>	Window size is not adjustable, it is impossible to switch to the full screen mode that allows for better experience with reading of logs in output area.
<i>Low</i>	Absence of help menu, FAQ or documentation.
<i>Low</i>	STOP and DELETE buttons are confusing, some users tried to stop the ARXML parsing with clicking on them.
<i>Low</i>	ID input field is confusing, during the rule insertion several users were stuck, attempting to enter id manually.
<i>Moderate</i>	Possibility of simultaneous processing of different ARXML files. It was identified, that user is allowed to select the second ARXML file while the first is still being processed.
<i>Low</i>	Some users were confused by the long processing of ARXML files, thinking that the GUI is stuck or does not respond.
<i>High</i>	While sending a JSON-RPC request, some users have faced the prolonged unresponsiveness of a user interface until the server feedback was received.
<i>High</i>	Sometimes, the inactive tab (signal modification / traffic filtering) was disappearing after maximizing of output area, requiring a repetitive resizing of output area to make it appear again.

■ **Table 5.4** User interface problems & defects identified by user-involved testing..

### 5.3.1.5 System Testing

As it was outlined in the test strategy (refer to section 5.2.1), the system testing was conducted simultaneously with the user-involved GUI testing. Manual analysis of the server-side logs has confirmed the successful reception of transmitted JSON-RPC requests, thereby categorizing the

<sup>4</sup>The term *bug* is controversial in this context, since the functionality was designed intentionally in the existing way and the software is still capable of fulfilling its intended functionality; thus, the identified issues may be considered as the design flaws.

test as successful. Although the scope of the testing conducted does not encompass the entirety of conceivable scenarios, it is deemed adequate to substantiate the functional capabilities of the prototype software.

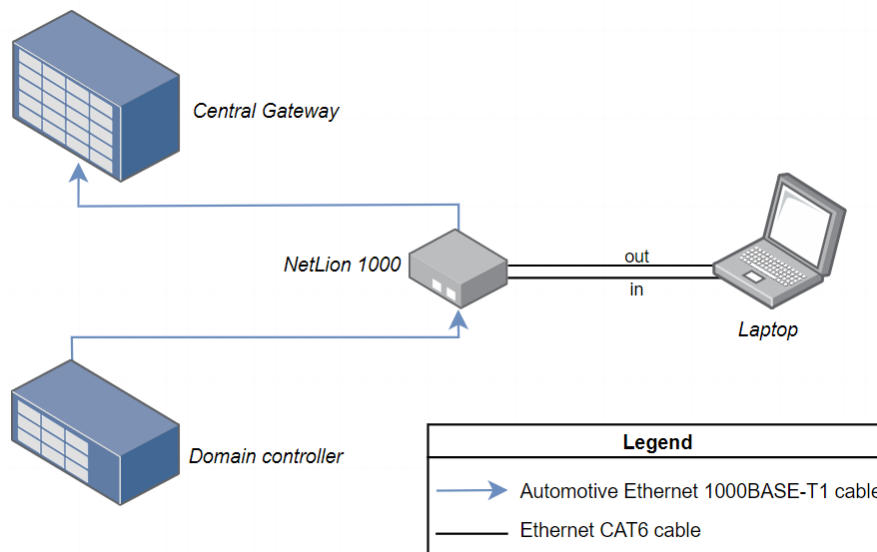
## 5.3.2 Server

### 5.3.2.1 System Testing: Setup of Environment

Testing was conducted in collaboration with Vehicle Testing department of Porsche Engineering Services s.r.o. and involved the use of a *simplified* and *specially created* simulation of a vehicle based on the VW E3 1.2 platform, utilizing dSPACE HIL (hardware-in-the-loop) technology. Moreover, it is worth mentioning that the simulation in use was *adjusted* to meet the *selected* functionality limitations of the developed system.

During the laboratory testing, the software-equipped laptops (Computers A & B) were alternately integrated into the in-vehicle communication network, connecting the ECU (HPCU) serving as a domain controller and the central gateway (refer to Figure 5.5 for details), connoting the use of domain-centralized E/E architecture in the simulated vehicle.

► Note 5.6. Importantly, the utilization of authentic Porsche vehicle simulations in the testing process, as well as the provision of further details regarding the used one, is restricted due to company policies on privacy & data protection.



■ **Figure 5.5** Testing setup.

Due to the infeasibility of directly interfacing the laptop with the typical Automotive Ethernet 1000BASE-T1 cable (refer to Figure 5.6), a NETLion 1000 Media Converter was employed, with the supplied voltage of 12 V.

The NETLion 1000 Media Converter is a development tool for 100/1000BASE-T1 networks aimed at logging and analysis of data traffic and/or conversion of 100/1000BASE-T1 Ethernet physical layers to 100BASE-TX/1000BASE-T Ethernet [81]. The very media converter is shown at Figure 5.7 and supports two operational modes. Namely, the *Dual Media Converter mode*, converting up to two 100/1000BASE-T1 streams to 100BASE-TX/1000BASE-T on two independent bi-directional converter channels, was utilized.



■ **Figure 5.6** Automotive Ethernet 1000BASE-T1 cable.



■ **Figure 5.7** NETLion 1000 Media Converter.

Importantly, since each of available laptops possessed only one Ethernet connector, the i-tec USB 3.0 Gigabit Ethernet Adapter was employed to establish an additional Ethernet connection via USB interface.

### 5.3.2.2 System Testing: Results

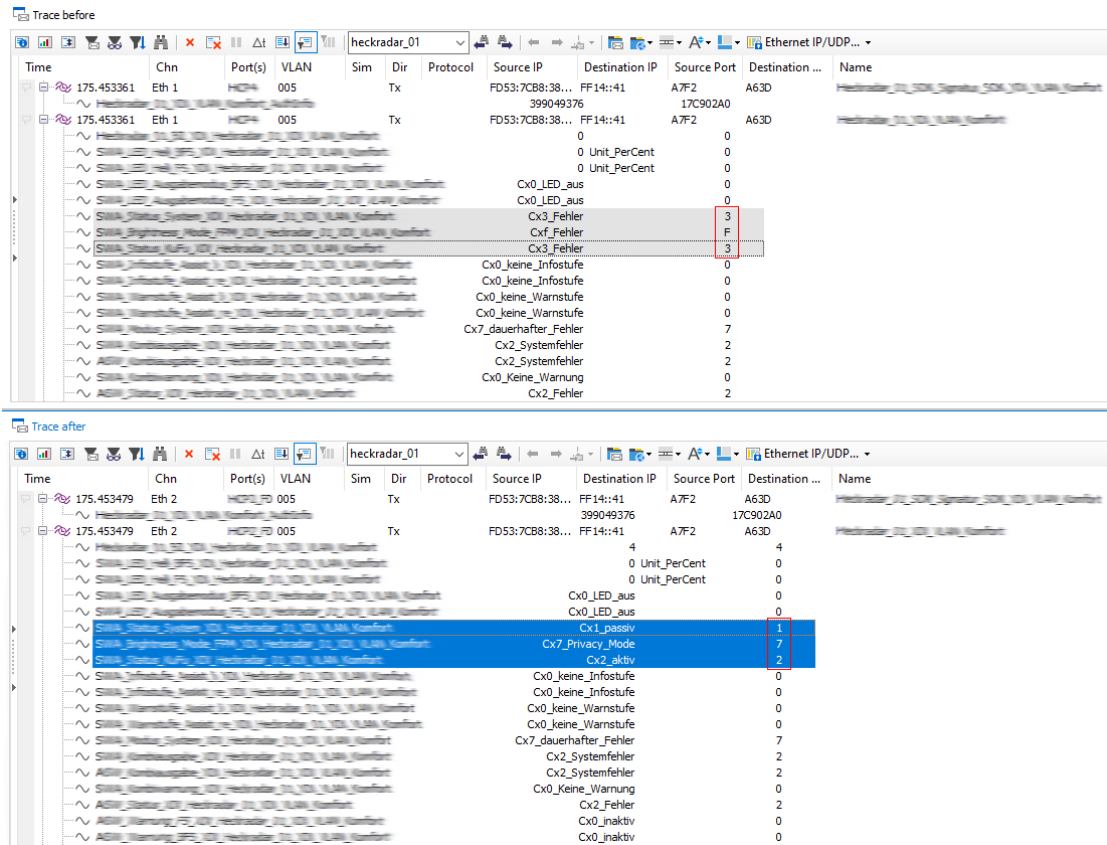
Prior to commencing the experimental procedures, an analysis of the existing average network load was conducted, corresponding to the data presented in Table 5.5.

Unfortunately, due to the lack of access to the configuration settings of the employed simulation and the impracticality of its rapid modification, it was not feasible to assess the software's performance under varying network loads during the testing process. This limitation consequently impacted the scope of the experiments conducted, disallowing the *needed* evaluation of server-side performance under the Class D network loads.

IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s
Master	2112,50	0	353,15	0
Slave	844,13	2113,03	97,97	361,24

■ **Table 5.5** Network interfaces throughput in the Transparent Gateway mode, one capture thread utilized.

► Note 5.7. Importantly, the utilized fragment of network topology denoted at Figure 5.5 implies the establishment of *unidirectional* connections from the Computer’s perspective. However, from the perspective of communicated parties, the connection remained *bidirectional*, since the authentic Automotive Ethernet network within the simulation experienced no prior adaptation to usage of unidirectional communication channels. In practise, this resulted into the repetitive attempts of central gateway to send the response data on the same channel backwards (domain controller was trying to receive these data), maintaining the original communication scenario and thus explaining the presence of incoming data on the slave interface. Despite this introduced some bias into the measurement, it has not significantly affected the assessment of traffic interference performed; the data incoming to the slave interface via communication bus were simply ignored.



■ **Figure 5.8** Verification of successfully performed manipulation with three signals<sup>5</sup> using CANoe.

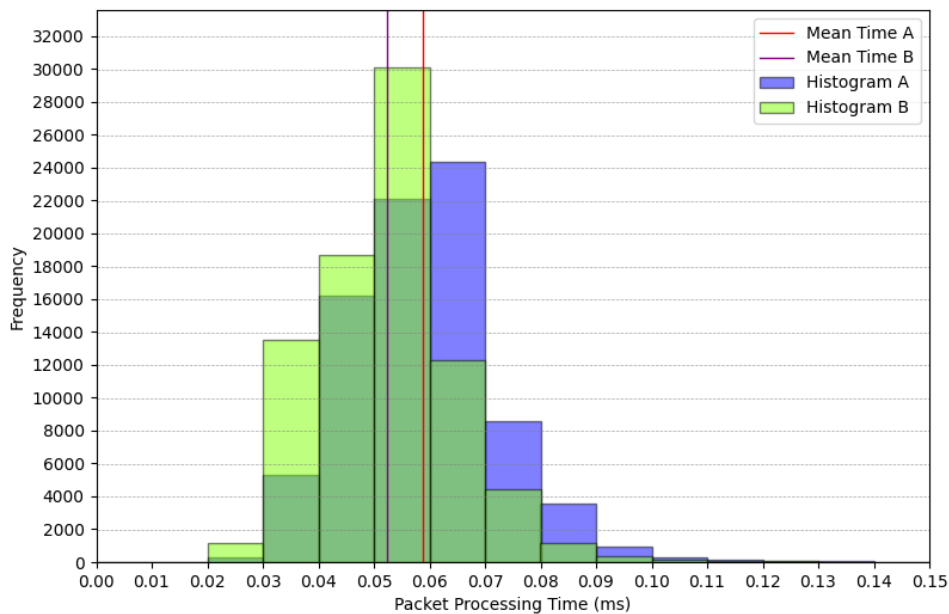
During the testing on the system level, *all* software use-case scenarios were *successfully* tested with both of available Computers in accordance with the derived testing strategy. The test results, verified both manually and automatically using CANoe traces (refer to Figure 5.8),

<sup>5</sup>Names of signals and PDUs are blurred due to the Porsche Engineering policy on privacy & data protection.

affirmed the developed software’s capability to interact with Automotive Ethernet traffic in the desired way.

**CPU Impact.** The initial observation made was that the task execution speed is indeed influenced by the qualitative characteristics of the processor. This was evidenced by the fact that the results obtained from measuring the average processing speed of a single packet (independently of the software use case) differed in a manner approximately equal to the ratio of the processors’ *maximal* performance ( $3.20/2.80 = 1.14$ , indicating *up to* 14% difference). To corroborate this, all subsequent results presented will concurrently display data obtained from both of Computers.

**Transparent Gateway.** The average calculated packet processing times in Transparent Gateway mode were found to be  $0.0598ms$  for Computer A and  $0.0532ms$  for Computer B (12% faster), as illustrated in Figure 5.9. This duration can be considered as the baseline processing overhead for transmitting an Ethernet packet, irrespective of its characteristics. Information regarding the average throughput of network interfaces is presented in Table 5.5.



■ **Figure 5.9** Average Ethernet packet processing time on Computers A and B in Transparent Gateway mode, one capture thread utilized.

During the analysis of the program’s performance in this usage scenario (as well as in all subsequent ones), certain artifacts were identified, the nature of which remains unclear. When determining the maximum packet processing time in this scenario, a solitary instance was recorded where the time exceeded 11 (!) milliseconds. Additionally, there were instances (up to 20 occurrences out of more than 90,000 transmitted packets) where the packet processing time ranged between 3 to 6 milliseconds.

Moreover, similar observations were acquired in during the testing of all of program use cases. Consequently, due to the sporadic nature of these data points, they were deemed outliers and excluded for simplicity of results interpretation, however, potentially deserving the further investigation.

Number of Threads	IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s
1	Master	2177,09	0,00	357,01	0,00
	Slave	842,59	1438,53	97,86	241,81
2	Master	2143,58	0	354,89	0
	Slave	844,13	1309,74	97,92	217,12
3	Master	2082,17	0	351,24	0
	Slave	833,28	1520,56	97,05	232,37
4	Master	2160,75	0	356,02	0
	Slave	836,94	1183,25	97,37	196,85

■ **Table 5.6** Registered throughput of network interfaces depending on the number of capture threads, having 19 active rules for signal modification applied in the same packet.

**Capture Threads.** Measurements on the average processing time conducted with varying numbers of threads (1, 2, 4) did not reveal significant differences in most of scenarios. It was noticed, that each thread *always* had merely one packet arrived at a time, conforming the low network utilization.

Only with the increased number of active rules (e.g.,  $> 20$ ), in some cases, the employment of additional thread(s) slightly benefited the overall throughput of network interfaces, comparing with the employment of a sole thread. In fact, *usually*, the utilization of additional threads (while having the same number of active rules) has led to the notable decline in the throughput of network interfaces, as seen in Table 5.6.

It is explained by the fact, that utilization of additional capture threads is only advantageous, having the high network load (packet processing time remains the same for each thread). In such cases, while the first thread is being busy with processing of a bunch of incoming packets (or one packet with multiple strategies respectively), the second thread would be able to capture newly arrived.

Given the average incoming bitrate of  $353kB/s$ , the utilization of additional threads with the small number of active rules (approximately  $< 20$ ) would deliver only the synchronization overhead.

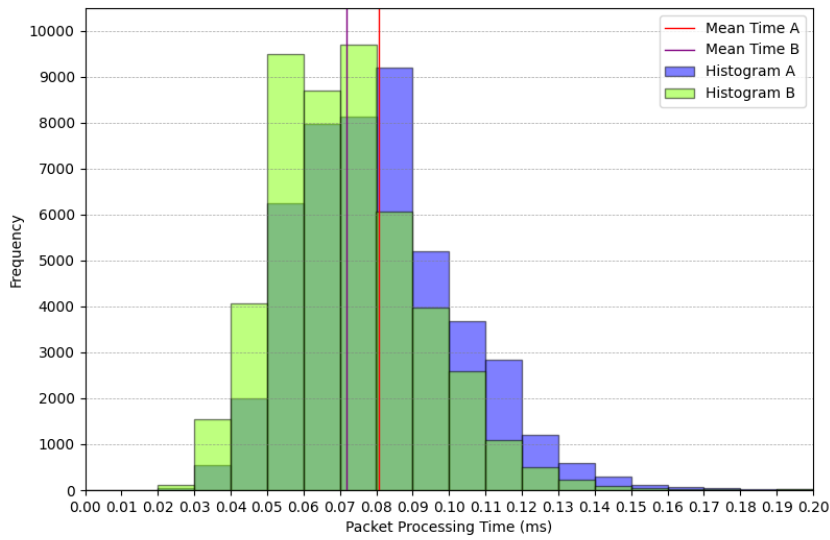
Additionally, the measurement with three capture threads led to discovery of strange persisting behavior of a program, sometimes having the average processing time almost 90% higher comparing to other number of threads (refer to Figures 5.14 and 5.15). The nature of this phenomenon remains *unknown*.

► Note 5.8. Importantly, the observed decline in throughput of network interfaces might be caused by further factors, not taken into account and requiring the further investigation.

**Other Use Cases.** Figure 5.10 denotes, that the time consumed for an exclusion of a single packet from traffic (traffic filtering) can be approximately up to  $0.18ms$ , with an average of  $0.0808ms$  for Computer A and  $0.0723ms$  for Computer B. Given the identified complexity of the packet parsing algorithm equal to  $O(NS)$  (refer to paragraph 4.3.3.6.2) and considering the deduced baseline processing overhead of  $0.0598ms$  (for Computer A), it can be claimed that, theoretically the exclusion of approximately *up to*<sup>6</sup> 140 packets from traffic by Computer A (if they arrive at once) will fit into ASIL FTTI set in the test strategy.

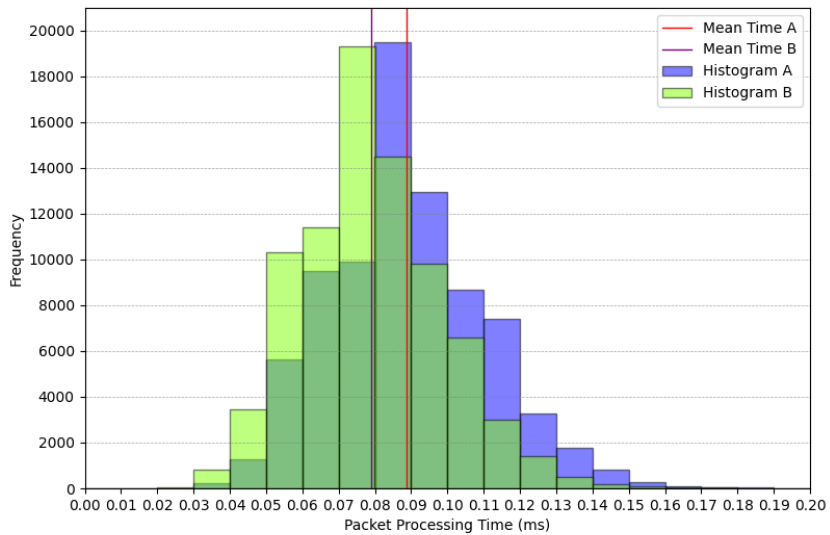
Although this scenario might seem to be practicality infeasible, similar calculations (it terms of maximal plausible number of signal modifications, etc.) could be easily performed and for further use cases if necessary, depending on the provided histograms (refer to Figures 5.11, 5.12 and 5.13).

<sup>6</sup>The maximal possible



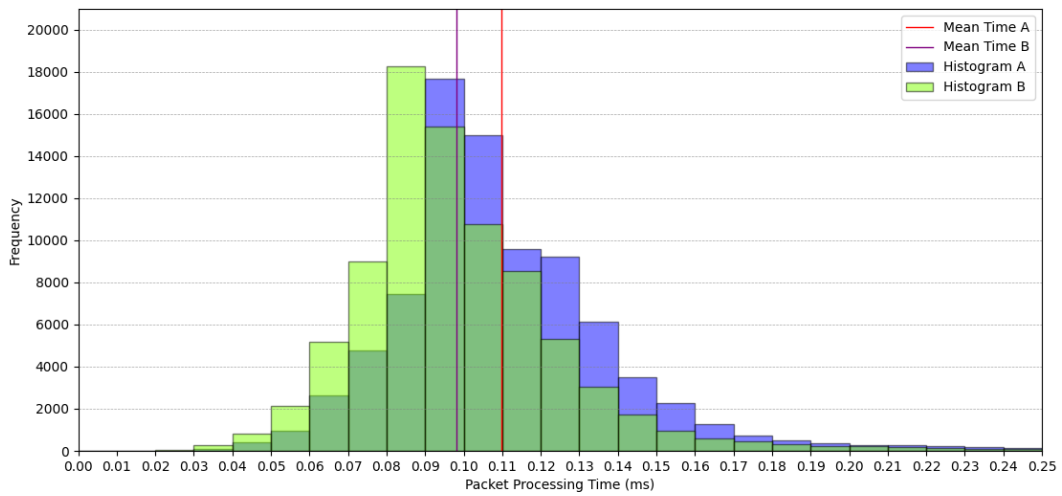
■ **Figure 5.10** Average Ethernet packet processing time on Computers A and B in Traffic Filtering mode, one capture thread utilized.

Importantly, at the time of testing, no impact of BZ auto-incrementation on the packet processing time was registered at all. Moreover, despite the successful testing of the BZ auto-incrementation functionality in line with the software’s use-case scenario, its practical application was rendered senseless due to the low overall network load, which resulted in only one packet being allocated to each capture thread.



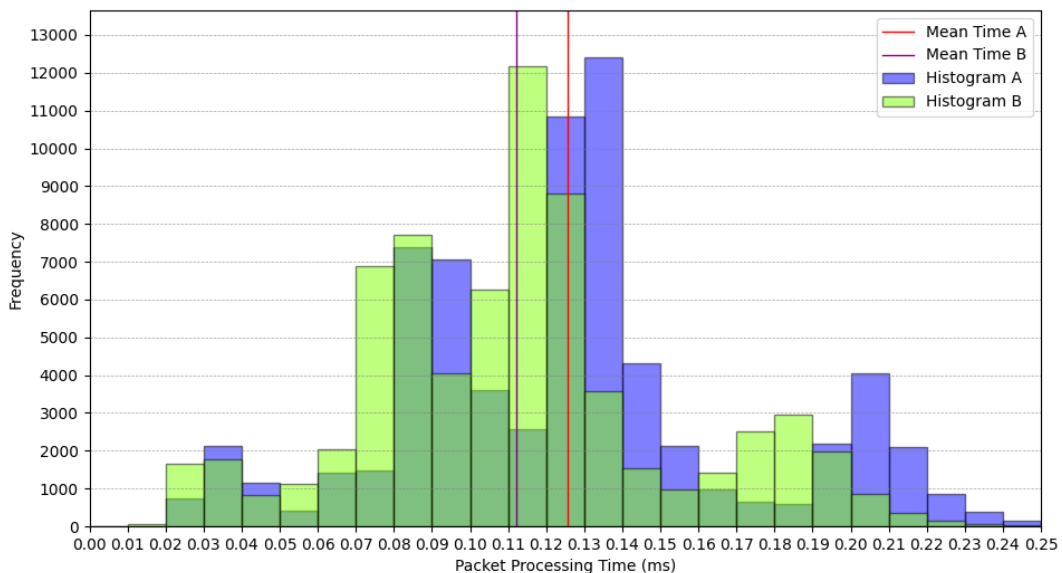
■ **Figure 5.11** Average Ethernet packet processing time on Computers A and B in Signal Modification mode (one signal modified), one capture thread utilized.





■ **Figure 5.12** Average Ethernet packet processing time on Computers A and B in Signal Modification mode (one signal modified) with in-PDU CRC recalculated, one capture thread utilized.

The subsequent Figures (namely 5.13, 5.14, 5.15 and 5.16) represent *combined* histograms of the obtained packet processing times. This implies that the average time depicted in the images is *illustrative* rather than definitive. It encompasses the parsing and modification (where necessary) of packets that are *non-compliant* (left part of the histogram), *partially compliant* (central part of the histogram), and *fully compliant* (rightmost part of the histogram) with the loaded rules.

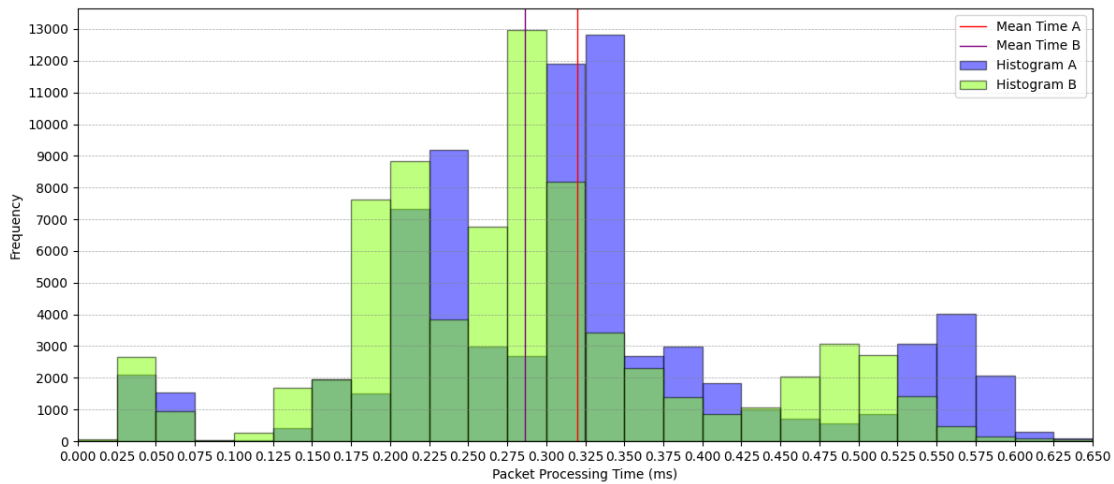


■ **Figure 5.13** Average Ethernet packet processing time on Computers A and B in Signal Modification mode with merely two in-PDU MACs recalculated, one capture thread utilized.

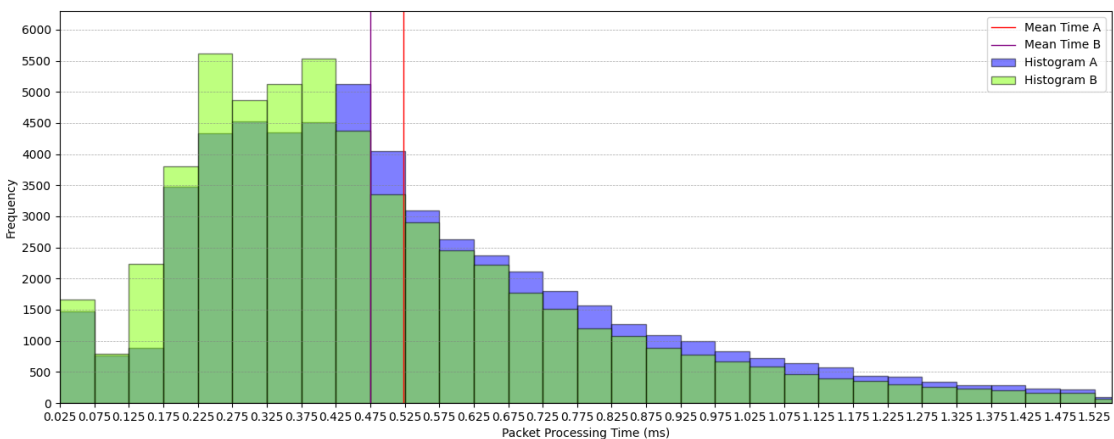


Such visualization was deemed appropriate as it allows for the approximate calculation of not only the average packet processing time based on its compliance with the rules, but also provides an intuitive understanding of the proportion of suitable packets among the total number processed.

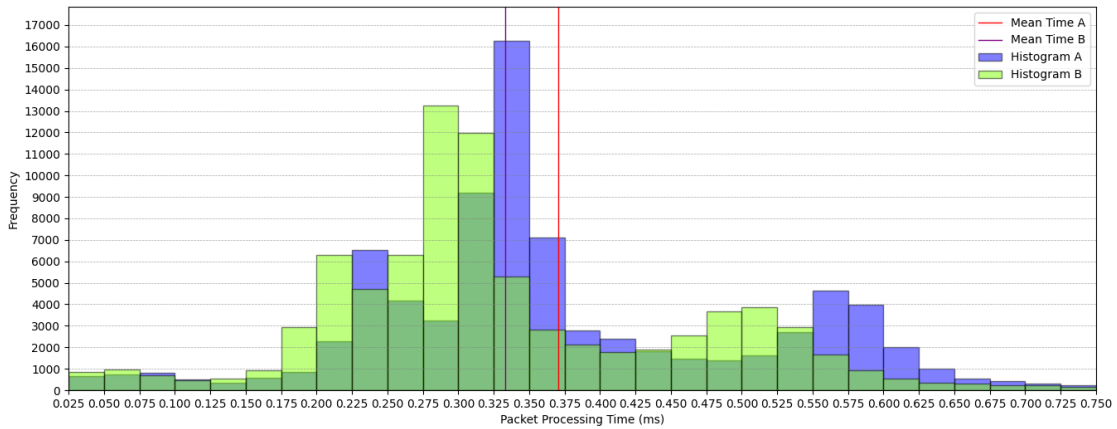
Additionally, the variability observed in all the results obtained can be attributed to factors such as the position of the targeted PDU within a specific packet (in scenarios of traffic interference use cases), as well as the potential variation in the processor’s operating frequency *during the testing process* (which was not measured). Consequently, the test results presented in this section should not be considered exhaustive or absolutely precise.



**Figure 5.14** Average Ethernet packet processing time on Computers A and B in Signal Modification mode (ten signals modified within the same PDU) with ten in-PDU CRC recalculated, one capture thread utilized.



**Figure 5.15** Average Ethernet packet processing time on Computers A and B in Signal Modification mode (ten signals modified within the same PDU), three capture thread utilized.



■ **Figure 5.16** Average Ethernet packet processing time on Computers A and B in Signal Modification mode (ten signals modified within the same PDU) with ten in-PDU CRC recalculated, one capture thread utilized.

The primary objective of the conducted testing was to evaluate the feasibility of signal manipulation within a set interval. This objective was achieved, characterizing the task as *feasible*. A more detailed analysis of this issue was deemed beyond the scope of this work due to its extensive nature.

## 5.4 Conclusion

This chapter has comprehensively and systematically addressed the challenges encountered in testing the multifaceted developed software, encompassing both theoretical foundations and practical applications for client and server system components.

Following a presentation of theoretical foundations of software testing and necessary definitions, a testing strategy was formulated, delineating the methods for verifying the stated functionality of the software. Given the status of the developed software as a functional prototype, the devised testing strategy was not exhaustive but rather focused on the functions and features deemed most critical. Specifically, emphasis was placed on the functionality and performance of the ARXML Parser abstract component, the usability of the GUI, and the overall functionality and performance of the server component.

The very testing was based predominantly on the assumption, largely correct, that other untested parts of the program were defect-free.

All individual steps, nuances, and observations were meticulously documented and are presented in the respective sections. Furthermore, identified defects were classified based on their severity. Among the most significant findings were:

1. The ARXML Parser demonstrated its ability to successfully extract necessary information required for signal modification & inherent security circumvention from available ARXML files. However, despite component tests confirming the capability to extract any ARXML data (in accordance with the schema and supported features), the implementation is deemed inefficient due to *extremely* excessive processing time for larger files or those containing more signals. Moreover, as it was proven, that the current implementation preclude the effective estimation of file processing time. Thus, in the context of the objectives set for this thesis, a sensible optimization for future developments could involve placing greater emphasis on

performance, by narrowing the scope of supported functions and changing the overall selected design approach.

2. The developed GUI interface and the client component (in general) are functional but contain some defects of varying severity related to design shortcomings, requiring further improvement.
3. The server component was tested under conditions, that were most feasible to attain within the constraints of this study, closely mirroring real-world scenarios for which it was designed. It experimentally confirmed the ability to modify signals in run-time in Automotive Ethernet networks. Additionally, implemented techniques for bypassing selected security mechanisms proved accurate and reliable. The *approximate* speeds of various traffic intervention scenarios were measured, indicating the feasibility of modifying multiple signals (not just one) considering different levels of protection within the timeframe established by ASIL FTTI.

It is important to note that due to limitations in the testing environment, the testing could not be conducted in full accordance with the provisions outlined in the testing strategy. Therefore, while the developed software potentially, based on some confirmed functionalities, can already assist in testing individual ECUs, it was not possible to assess its performance under varying network loads. Moreover, during the testing process, several unexpected artifacts of unknown nature were identified. This indicates imperfections in the testing strategy and gaps in the collected technical and theoretical knowledge, given the lack of adequate explanations. This carries certain risks for its real-world application as-is and implies a need for further research in this area.

In summarizing the results of the conducted testing, it can be confidently stated that the developed software, despite some shortcomings, has demonstrated the *feasibility* of successful and *unobtrusive* intervention in the traffic of the Automotive Ethernet. This effectively proves its compliance with both the functional and non-functional requirements set forth.



# Conclusion

This thesis was set to explore the intricate relationship between automotive networking technologies, vehicular electronic and electrical architectures, and the nuances of driving automation. Furthermore, the main objective of this work was to develop an innovative prototype system designed to facilitate the testing of individual electronic control units at the system level via unobtrusive signal manipulation, bypassing security mechanisms set by functional safety standards. Unfortunately, during the course of this research, devices with similar functionalities became available in the market, significantly diminishing the initially anticipated innovation of the results.

This research evolved progressively, with each subsequent part of the work building on the foundations laid by the previous chapters. The detailed results of each stage of this study were documented in the closing sections of corresponding chapters.

The foundation of this work was laid in Chapter 1, outlining the primary objectives, methodology, and sequence of this research.

Chapter 2 delves deep into the realm of driving automation, dissecting the taxonomy of driving automation levels, the intricacies of functional safety, and the complex E/E architecture of vehicles. The role of AUTOSAR in vehicle communication and the security aspects of automotive networking were also examined, setting a solid theoretical foundation for the practical components of this study. A crucial component of this section was the analysis of existing protection mechanisms in Automotive Ethernet networks, which allowed for the synthesis of methods to circumvent some of these mechanisms.

The focus of Chapter 3 was on synthesizing requirements for the proposed software system. This involved a careful consideration of functionality limitations based on the conclusions drawn from the theoretical part of the study, implicitly indicating the achievable functionality of the final system within the scope of this work. User expectations, and both hardware and software requirements were consulted and derived in collaboration with leading automotive industry experts from Porsche Engineering. This rigorous approach ensured that the developed system was grounded in real-world applicability and industry relevance.

Chapter 4 presents a technical monograph that describes every step undertaken in the design and implementation of the software system. The architectural design, choice of technologies and libraries, and a detailed insight into the system's implementation were discussed. A reasoned decision was made in favor of a client-server architecture, which allowed for the partitioning of the necessary functionality into smaller, more manageable tasks, as well as their independent implementation & testing. The chapter concluded with an evaluation of how the design and implementation met the set requirements, along with an analysis of the benefits and drawbacks of the implementation.

Chapter 5 focuses on verifying the developed software's compliance with the derived functional

and non-functional requirements. The theoretical aspects of software testing were discussed, and a testing strategy was formulated, centering the forthcoming tests on the most critical aspects of the final system. The chapter described the testing setup and presented the verification and validation of the system's functionality through various tests. Importantly, the functionality simulation-based testing was conducted with support of Porsche Engineering in controlled environments, under conditions closely resembling real-world scenarios.

Some tests led to the identification of several software defects, while others proved the compliance of the resulting system with the initial requirements. During the testing process, the capability of the developed software to perform signal manipulation on the Automotive Ethernet, invisible to the communicating parties, by circumventing selected security mechanisms was documented.

In conclusion, it is possible to claim that the thesis has successfully achieved all the objectives set. Moreover, this thesis opens several avenues for further research, dictated by the results obtained. The exploration of additional possibilities of data manipulation within the Automotive Ethernet network, the refinement of the software system to enhance its efficiency and scalability, and the potential application of these technologies in vehicle testing present new opportunities for continued innovation.

In conclusion, this master thesis could stand as a *notable* theoretical and practical contribution to the field of automotive technologies, serving as a specimen for the development and testing of similar systems.

# Bibliography

1. ORGANISATION INTERNATIONALE DES CONSTRUCTEURS D'AUTOMOBILES – OICA. *World Motor Vehicle Production* [<https://www.oica.net/category/production-statistics/>]. 2023.
2. MATHEUS, Kirsten; KÖNIGSEDER, Thomas. *Automotive Ethernet*. 3rd ed. Cambridge University Press, 2021. Available from DOI: 10.1017/9781108895248.
3. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. SAE International, 2018. No. J3016-201806. Available also from: [https://www.sae.org/standards/content/j3016\\_201806/](https://www.sae.org/standards/content/j3016_201806/).
4. XIE, S.; HU, J.; BHOWMICK, P.; DING, Z.; ARVIN, F. Distributed Motion Planning for Safe Autonomous Vehicle Overtaking via Artificial Potential Field. *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*. 2022, vol. 23, no. 11. ISSN 1524-9050. Available also from: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9830995>.
5. XIE, S.; HU, J.; DING, Z.; ARVIN, F. Cooperative Adaptive Cruise Control for Connected Autonomous Vehicles using Spring Damping Energy Model. *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*. 2023, vol. 72, no. 3. ISSN 1524-9050. Available also from: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9933795>.
6. ATAKISHIYEV, Shahin; SALAMEH, Mohammad; YAO, Hengshuai; GOEBEL, Randy. *Explainable Artificial Intelligence for Autonomous Driving: A Comprehensive Overview and Field Guide for Future Research Directions*. 2023. Available from arXiv: 2112.11561 [cs.AI].
7. SERBAN, Alex; POLL, Erik; VISSER, Joost. A Standard Driven Software Architecture for Fully Autonomous Vehicles. *Journal of Automotive Software Engineering*. 2020, vol. 1, pp. 20–33. ISSN 2589-2258. Available from DOI: <https://doi.org/10.2991/jase.d.200212.001>.
8. PARET, Dominique; REBAINE, Hassina; ENGEL, B. A. DAS, ADAS, HADAS, and AVs – L3, L4, L5! In: *Autonomous and Connected Vehicles: Network Architectures from Legacy Networks to Automotive Ethernet*. 2022. Available from DOI: 10.1002/9781119816140.ch3.
9. PROJECT, eSafety. *Preventive and Active Safety Applications Integrated Project Contract number FP6-507075 eSafety for road and air transport Code of Practice for the Design and Evaluation of ADAS*. Version 5.0. eSafety for road and air transport, 2009.
10. DAS, A.; KUMAR, A.; VEERAVALLI, B. Communication and migration energy aware task mapping for reliable multiprocessor systems. *Future Gener. Comput. Syst.* 2014, vol. 30, pp. 216–228. Available from DOI: 10.1016/j.future.2013.09.011.

11. *ISO 26262-4:2018(E) – Road vehicles – Functional safety – Parts 1-12*. Geneva: International Organization for Standardization (ISO), 2018. E. Available also from: <https://www.iso.org/standard/68383.html>.
12. *IEC 61508 Ed. 2.0: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. International Electrotechnical Commission, 2010.
13. INSTRUMENTS, National. *Understanding the ISO 26262 Functional Safety Standard*. 2023. Available also from: <http://www.ni.com/white-paper/13647/en/#toc2>.
14. *ISO 26262: Road vehicles – Functional safety*. International Organization for Standardization, 2018. Available from ISO, <https://www.iso.org/standard/68383.html>.
15. SHIBAHARA, Shinichi. Functional safety SoC for autonomous driving. In: *2018 IEEE Custom Integrated Circuits Conference (CICC)*. 2018, pp. 1–8. Available from DOI: 10.1109/CICC.2018.8357065.
16. BIEMAN, James M.; DREILINGER, Daniel; LIN, Linda. Using Fault Injection to Increase Software Test Coverage. In: *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*. 1996, pp. 166–174. Available from DOI: 10.1109/ISSRE.1996.558776.
17. *ISO/PAS 21448:2019 Road Vehicles – Safety of the Intended Functionality*. International Organization for Standardization, 2019.
18. BANDUR, V.; SELIM, G.; PANTELIC, V.; LAWFORDE, M. Making the Case for Centralized Automotive E/E Architectures. *IEEE Trans. Veh. Technol.* 2021, vol. 70, pp. 1230–1245.
19. NAVET, Nicolas; SIMONOT-LION, Françoise. Automotive Embedded Systems Handbook. In: 2017. Available also from: <https://api.semanticscholar.org/CorpusID:116577082>.
20. MORRIS, Brendan. *E/E Architecture Considerations for AV Development*. Siemens Digital Industries Software, 2021-03. Available also from: <https://www.eetimes.com/e-e-architecture-considerations-for-av-development/>.
21. PELLICCIONE, P.; KNAUSS, E.; HELDAL, R.; ÅGREN, S.M.; MALLOZZI, P.; ALMINGER, A.; BORGENTUN, D. Automotive Architecture Framework: The Experience of Volvo Cars. *J. Syst. Archit.* 2017, vol. 77, pp. 83–100.
22. ASKARIPOOR, Hadi; HASHEMI FARZANEH, Morteza; KNOLL, Alois. E/E Architecture Synthesis: Challenges and Technologies. *Electronics*. 2022, vol. 11, no. 4. ISSN 2079-9292. Available from DOI: 10.3390/electronics11040518.
23. BUTZKAMM, Cornelius; BRAND, Konstantin. E/E Architecture in the HARRI Innovation Platform. *ATZelectronics worldwide*. 2020, vol. 15, no. 3, pp. 18–24. ISSN 2524-8804. Available from DOI: 10.1007/s38314-019-0160-z.
24. ASKARIPOOR, Hadi; FARZANEH, Morteza Hashemi; KNOLL, Alois. A Platform to Configure and Monitor Safety-Critical Applications for Automotive Central Computers. In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2021, pp. 1–4. Available from DOI: 10.1109/ETFA45728.2021.9613692.
25. MEHRARA, M.; JABLIN, T.; UPTON, D.; AUGUST, D.; HAZELWOOD, K.; MAHLKE, S. Multicore Compilation Strategies and Challenges. *IEEE Signal Process. Mag.* 2009, vol. 26, pp. 55–63.
26. ASKARIPOOR, H.; FARZANEH, M.H.; KNOLL, A. A Platform to Configure and Monitor Safety-Critical Applications for Automotive Central Computers. In: *Proceedings of the 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vasteras, Sweden, 2021, pp. 1–4.



27. ZERFOWSKI, D.; LOCK, A. Functional Architecture and E/E-Architecture—A Challenge for the Automotive Industry. In: *Internationales Stuttgarter Symposium*. Berlin/Heidelberg, Germany: Springer, 2019, pp. 909–920.
28. APOSTU, S.; BURKACKY, O.; DEICHMANN, J.; DOLL, G. *Automotive Software and Electrical/Electronic Architecture: Implications for OEMs*. 2019.
29. HOLMES, J.H. Freddie; MORETON, J. *Zonal E/E Architectures: The Cornerstone of Future Mobility Development*. 2021. Available also from: <https://www.automotiveworld.com/articles/zonal-e-e-architectures-the-cornerstone-of-future-mobility-development>.
30. JIANG, S. *Vehicle E/E Architecture and Its Adaptation to New Technical Trends*. Warrendale, PA, USA, 2019. Tech. rep. SAE Technical Paper.
31. SHAVIT, M.; GRYC, A.; MIUCIC, R. *Firmware Update over the Air (FOTA) for Automotive Industry*. Warrendale, PA, USA, 2007. Tech. rep. SAE Technical Paper.
32. CORRIGAN, Steve. *Introduction to the Controller Area Network (CAN)*. s.l., 2008. Tech. rep. Texas Instruments.
33. *Class A Application Definition (J2507-1)*. SAE, 2008. J2507-1.
34. WANG, Yunpeng; TIAN, Daxin; SHENG, Zhengguo; JIAN, Wang. *Connected Vehicle Systems: Communication, Data, and Control*. 1st ed. CRC Press, 2020.
35. LEEN, Gabriel; HEFFERNAN, D.; DUNNE, A. Digital networks in the automotive vehicle. *Computing & Control Engineering Journal*. 2000, vol. 10, pp. 257–266. Available from DOI: 10.1049/ccej:19990604.
36. REIF, Prof. Dr.-Ing. Konrad (ed.). *Automotive Mechatronics: Automotive Networking, Driving Stability Systems, Electronics*. Bosch Professional Automotive Information, Springer, 2014. ISBN 978-3-658-03974-5 (Print), ISBN 978-3-658-03975-2 (eBook). Available from DOI: 10.1007/978-3-658-03975-2.
37. *Road Vehicles - Controller Area Network (CAN) - Part 5: High-speed Medium Access Unit with Low-power Mode*. 2007. Tech. rep. International Organization for Standardization (ISO).
38. MOST COOPERATION. *MOST Specification Rev. 3.0 E2*. 2010. Available also from: <http://www.mostcooperation.com/publications/specifications-organizational-procedures/>.
39. MASLOUH, J.; ERRAMI, A.; KHALDOUN, M. Resolving the Access Conflict for Shared Ethernet Communication Channel. In: *IEEE International Conference on Next Generation Networks and Services*. University of Oxford, 2014, pp. 80–87.
40. *ISO/IEC 7498-1:1994 - Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model* [<https://www.iso.org/standard/20269.html>]. International Organization for Standardization, 1994.
41. *ISO 7498:1984 - Information processing systems — Open Systems Interconnection — Basic Reference Model* [<https://www.iso.org/standard/14252.html>]. International Organization for Standardization, 1984.
42. *ISO 7498:1984/Cor 1:1988 - Information processing systems — Open Systems Interconnection — Basic Reference Model — Technical Corrigendum 1* [<https://www.iso.org/standard/14253.html>]. International Organization for Standardization, 1988.
43. POSTEL, Jon. *Request for Comments 768 - User Datagram Protocol (UDP)*. Marina del Rey, USA: Information Science Institute, University of Southern California, 1980.
44. STARON, Mirosław. *Automotive Software Architectures: An Introduction*. Gothenburg, Sweden: Springer, 2017. ISBN 978-3-319-58609-0. Available from DOI: 10.1007/978-3-319-58610-6.

45. AUTOSAR. *AUTOSAR Adaptive Platform for Connected and Autonomous Vehicles*. 2016. Available also from: [www.autosar.org](http://www.autosar.org).
46. ISO 17356-4:2005 *Road vehicles—Open interface for embedded automotive applications – Part 4: OSEK/VDX Communication (COM)* [<https://www.iso.org/standard/38884.html>]. International Organization for Standardization, 2005.
47. AUTOSAR. *Requirements on Communication*. 2021. AUTOSAR Standard, Document Identification No 2. AUTOSAR. Classic Platform, Part of Standard Release R21-11, Document Status: Published.
48. AUTOSARGBR. *Specification of PDU Router*. 2008-08. Technical Report.
49. AUTOSAR. *Specification on SOME/IP Transport Protocol* [<http://some-ip.com/standards.shtml>]. 2022. AUTOSAR Release 22-11, CP, No.809.
50. AUTOSAR. *Specification of TCP/IP Stack*. 2017. AUTOSAR Standard, Document Identification No 617. AUTOSAR. Classic Platform, Part of Standard Release 4.3.1, Document Status: Final.
51. AUTOSAR. *ARXML Serialization Rules*. 2022.
52. AUTOSAR. *Application Interfaces User Guide*. 2022.
53. CAESAR, Gaius Julius. *De Bello Gallico*. circa 50-44 BCE. Available online at <http://www.thelatinlibrary.com/caesar/gall1.shtml>.
54. IEEE 802.1 WORKING GROUP. *802.1AE - Media Access Control (MAC) Security* [<https://www.ieee802.org/1/files/public/docs2018/new-802-1ae-2018.pdf>]. 2018.
55. LINDNER, M. *Security Architecture for IP (IPsec)* [[www.ict.tuwien.ac.at/lva/384.081/infobase/L97-IPsec\\_v4-7.pdf](http://www.ict.tuwien.ac.at/lva/384.081/infobase/L97-IPsec_v4-7.pdf)]. 2007.
56. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3* [RFC 8446]. RFC Editor, 2018. Request for Comments, no. 8446. Available from DOI: 10.17487/RFC8446.
57. RESCORLA, E.; MODADUGU, N. *Datagram Transport Layer Security* [RFC 4347]. RFC Editor, 2006. Request for Comments, no. 4347. Available also from: <https://datatracker.ietf.org/doc/html/rfc4347>. Historic (changed from Proposed Standard January 2021).
58. AUTOSAR. *Specification of Secure Onboard Communication Protocol* [Initial release]. 2020-11. AUTOSAR Documentation, 969. AUTOSAR. Available also from: <https://www.autosar.org/standards/foundation/>. Part of AUTOSAR Standard Foundation, Release R20-11.
59. GRÜMER, Patrick Alexandre Almeida. *Attack Model Implementation for a Secure Onboard Communication from an Automotive ECU*. 2019. Mestrado em Segurança Informática. Faculdade de Ciências da Universidade do Porto.
60. ASAM. *FlexDevice* [<https://www.asam.net>]. [N.d.].
61. STAR ELECTRONICS. *FL3X Switch 1000BASE-T1* [<https://flex-product.com>]. [N.d.].
62. STAR ELECTRONICS. *FL3X Device-L<sup>2</sup> for complex networking technology* [<https://flex-product.com>]. [N.d.].
63. STAR ELECTRONICS GMBH & CO. KG. *FlexConfig RBS supports all conventional bus systems* [<https://www.asam.net>]. [N.d.].
64. MANNAERT, H.; VERELST, J.; DE BRUYN, P. *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. nsi-Press powered bei Koppa, 2016. ISBN 9789077160091. Available also from: [https://books.google.cz/books?id=0rA\\_tAEACAAJ](https://books.google.cz/books?id=0rA_tAEACAAJ).

65. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Professional, 1995.
66. SEDLAKOVA, J; DANIORE, P; HORN WINTSCH, A; WOLF, M; STANIKIC, M; HAAG, C, et al. Challenges and best practices for digital unstructured data enrichment in health research: A systematic narrative review. *PLOS Digital Health*. 2023, vol. 2, no. 10, e0000347. Available from DOI: [10.1371/journal.pdig.0000347](https://doi.org/10.1371/journal.pdig.0000347).
67. RIVERBANK COMPUTING LIMITED. *PyQt6 Documentation*. 2023. Available also from: <https://www.riverbankcomputing.com/static/Docs/PyQt6/>.
68. MEYER, Bertrand. *Object-Oriented Software Construction (2nd Ed.)* New York, NY, USA: Prentice-Hall, Inc., 1997. ISBN 0136291554.
69. MARTIN, Robert C. Object Oriented Design Quality Metrics: An Analysis of Dependencies. *C++ Report*. 1995.
70. STANDARDS, National Institute of; (NIST), Technology. *Advanced Encryption Standard (AES)*. NIST, 2001. Available also from: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>. Federal Information Processing Standards Publication 197.
71. DAEMEN, Joan; RIJMEN, Vincent. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 978-3-540-42580-9.
72. BERTONI, Guido; BREVEGLIERI, Luca; FRAGNETO, Pasqualina; MACCHETTI, Marco; MARCHESIN, Stefano. Efficient software implementation of AES on 32-bit platforms. In: JR., Burton S. Kaliski; KOÇ, Çetin Kaya; PAAR, Christof (eds.). *CHES 2002: Cryptographic Hardware and Embedded Systems*. Springer Berlin Heidelberg, 2002, vol. 2523, pp. 159–171. Lecture Notes in Computer Science. ISBN 978-3-540-00409-3. Available from DOI: [10.1007/3-540-36400-5\\_12](https://doi.org/10.1007/3-540-36400-5_12).
73. MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN 978-0-8493-8523-0.
74. CRYPTON, Alex. *Advanced Encryption Standard Explained*. New York: Cryptography Press, 2023.
75. STALLINGS, William. *Cryptography and Network Security: Principles and Practice*. 7th. Pearson, 2016. ISBN 978-0-13-444428-4.
76. DWORKIN, M. *The AES-CMAC Algorithm [RFC 4493]*. Network Working Group, 2006. Available also from: <https://www.rfc-editor.org/rfc/rfc4493.txt>.
77. BLACK, Rex; VEENENDAAL, Erik Van; GRAHAM, Dorothy. *Foundations of Software Testing – ISTQB® Certification*. 3rd ed. Cengage Learning, 2012.
78. *Automotive SPICE*. .13rd ed. Automotive Special Interest Group, 2005. Available also from: <https://vda-qmc.de/en/automotive-spice/>. Based on ISO/IEC 330xx series. Used for evaluating development processes in the automotive industry.
79. PAVLICEK, Josef. *User Interface Testing* [<https://docs.google.com/presentation/d/1t4kCvHJSqpzqff30JhoAzPvaJQQELJ990geai3K9e8/edit#slide=id.p>]. 2023. [Online].
80. MICROSOFT. *Windows User Experience Guidelines*. 2023. Available also from: <https://learn.microsoft.com/en-us/windows/win32/uxguide/guidelines>.
81. B-PLUS GMBH. *NETLion1000 Data Sheet* [[https://www.b-plus.com/fileadmin/data\\_storage/Data\\_Storage/Produkte/Produktdatenblaetter\\_AE/Datasheet\\_NETLion1000\\_en\\_1.1.pdf](https://www.b-plus.com/fileadmin/data_storage/Data_Storage/Produkte/Produktdatenblaetter_AE/Datasheet_NETLion1000_en_1.1.pdf)]. 2020. Version 1.2.



# Attachments

aepril-client.....	client-side of the developed system
├─ arxml	
├─ common	
├─ config	
├─ logic	
├─ views	
├─ app.py	
├─ requirements.txt	
aepril-server .....	server-side of the developed system
├─ conf	
├─ db	
├─ src	
├─ .gitmodules .2 build.sh	
├─ CMakeLists.txt	
├─ FindPCAP.CMake	
├─ README.md	
├─ setup.sh	
thesis .....	thesis
├─ text.....	text of the thesis in L <sup>A</sup> T <sub>E</sub> X
├─ thesis.pdf.....	text of the thesis in PDF