POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A Service-Oriented Design Framework for Automotive Architectures

Supervisor
Prof. Fulvio RISSO

Candidate
Giorgia TORTORELLI

Company Supervisors
Ing. Paolo PICCIAFOCO
Ing. Massimiliano RASPANTE
Ing. Domenico PERRONI

Anno accademico 2024-2025

Summary

In recent years, the automotive industry has undergone a profound transformation, driven by the growing demand for advanced functionalities such as autonomous driving, advanced driver assistance systems (ADAS), and sophisticated infotainment platforms. These innovations have led to a significant increase in the complexity of vehicle electronics, challenging the limits of traditional architectures based on numerous Electronic Control Units (ECUs), each dedicated to a specific function (like braking or managing the engine) and interconnected through legacy protocols like CAN or LIN.

This complexity has highlighted the limitations of static and tightly coupled systems, making it increasingly difficult to integrate new features in a modular and scalable way. To address these challenges, the industry is gradually shifting toward **Service-Oriented Architectures (SOA)**, in which software services are designed to be **independent**, **interoperable**, and **dynamically deployable**, enabled by the adoption of standardized **interfaces**. This paradigm, supported by modern communication protocols such as **SOME/IP over Ethernet**, promotes flexibility, reusability, and scalability.

This thesis, carried out in collaboration with **Italdesign**, explores the transition toward SOA in the automotive domain, proposing an innovative and distributed approach to service management, and also streamlining the path from system modeling to the development and validation of a working application. The proposed methodology aims to reduce production times, which are currently long and fragmented due to the involvement of multiple stakeholders (OEMs and Tier-1s), making it possible to quickly generate testable prototypes directly from the architectural model.

Contents

List of Tables 7

List of Figures 8

1 Introduction 11

2 State of the art 14

3 Background and Technological Principles 18
- 3.1 Service-Oriented architecture vs Software-Oriented architecture . . 18
  - 3.1.1 Software-Oriented architecture: a monolithic approach . . . 18
  - 3.1.2 Service-Oriented architecture: modularity and flexibility . . 19
  - 3.1.3 Comparing traditional and SOA approaches 21
- 3.2 Zonal architecture: evolution of E/E 24
  - 3.2.1 Overcoming the decentralized model 24
- 3.3 The V-Model 26
- 3.4 Model-Based Systems Engineering in the automotive sector 29

4 Tools and Technologies for SOA Automotive Development 32
- 4.1 PREEvision: a model-based tool for E/E design 32
  - 4.1.1 Support for AUTOSAR classic and adaptive 34
  - 4.1.2 Advantages in the automotive sector 35
- 4.2 SOME/IP protocol: middleware for Service-Oriented architectures . 36

5 Modeling a Service-Oriented Architecture with PREEvision 44
- 5.1 Adopted development approach 44
- 5.2 Modeling in PREEvision 45
- 5.3 Requirements definition 46
- 5.4 Software architecture 49
- 5.5 Service architecture 53
  - 5.5.1 Custom attributes definition and dynamic service shifting . . 60
- 5.6 Hardware architecture 61
  - 5.6.1 Mapping software/hardware layers 64
- 5.7 Communication layer 66

6 Building C++ application using SOME/IP 72
6.1 Parsing the ARXML file and generating the JSON file 73
6.2 Generating configuration files for vSomeIP 75
6.3 Implementation of the C++ application with vSomeIP 77
6.3.1 Provider implementation 79
6.3.2 Consumer implementation 80

7 Conclusions 84
7.1 Main outcomes 84
7.2 Limitations and improvements 86
7.3 Future work 86

Bibliography 88

List of Tables

5.1 Example of Requirements for Alternative Route Service 47

List of Figures

3.1 Service-Oriented architecture 20
3.2 Monolith vs microservices [22] 22
3.3 Comparison between Software-Oriented and Service-Oriented architectures 24
3.4 Zonal topology [20] 26
3.5 V-Model 27

4.1 PREEvision layers [15] 33
4.2 AUTOSAR adaptive [21] 35
4.3 Typical use cases of the SOME/IP protocol in automotive systems [18] 36
4.4 How SOME/IP works on Ethernet backbone [18] 37
4.5 Request/Response 38
4.6 Fire-and-Forget 38
4.7 Event 39
4.8 Field 39
4.9 SOME/IP header format [23] 41
4.10 SOME/IP-SD header format [24] 42

5.1 Project workflow 45
5.2 System and software design process in PREEvision [15] 50
5.3 Alternative route service software diagram 52
5.4 Alternative route service service diagram 54
5.5 Requirement GPS mapping 55
5.6 Service-Oriented modeling workflow in PREEvision [15] 56
5.7 Service interfaces 57
5.8 Software design SOA 58
5.9 Custom attributes 60
5.10 Get zonal info service 61
5.11 Hardware Architecture 62
5.12 Mapping software architecture 65
5.13 Mapping service architecture 66
5.14 Signal mapping between infotainment and front right zone ECU . . 68
5.15 ECU Ethernet configuration 69
5.16 Socket configuration in the Ethernet network 69
5.17 Mapping between service instances provided and consumed 70

5.18 Mapping of event groups between providers and consumers 71

6.1 SOME/IP services 73
6.2 Service deployments 74
6.3 Configuration file 76
6.4 vSomeIP overview 78
6.5 Provider sequence diagram 81
6.6 Consumer sequence diagram 83

Chapter 1

Introduction

In recent years, the evolution of digital technologies has radically transformed the way complex systems are designed and implemented. Among the sectors most affected by this revolution is the **automotive industry**, which has seen a clear shift from traditional mechatronic paradigms towards increasingly **software-defined architectures**. Modern vehicles are no longer simple modes of transportation, but sophisticated cyber-physical systems equipped with connectivity, advanced computational capabilities, and intelligent functions.

In particular, the automotive industry is currently at the center of a deep digital transformation, driven by the introduction of advanced features such as **autonomous driving**, **advanced driver assistance systems** (ADAS), **Vehicle-to-Everything** (V2X) connectivity and **electric vehicles**. This scenario has put traditional electronic architectures in crisis, as they are based on separate functional control units and have obvious limitations in terms of scalability, interoperability, and efficient resource management.

To address these challenges, the industry is gradually moving towards more flexible and modular architectural solutions, such as **Service-Oriented architectures** (SOA) and **zonal architectures**. SOA allows software functionality to be decoupled from physical hardware, promoting reusability, updatability, and dynamic service management. Zonal architecture, on the other hand, logically divides the vehicle into physical zones, each controlled by a central node, simplifying the electrical topology and reducing wiring complexity.

This is the context for the work described in this thesis, whose goal is to design and validate a service-oriented architecture applied to a zonal hardware infrastructure. A further goal is to create an **internal benchmark** to demonstrate how the initial phases of the automotive software development cycle can be significantly optimized. The proposed approach aims to partially automate the design flow, enabling a fluid transition from abstract modeling to concrete implementation, with a consequent reduction in errors and increased consistency between phases.

The project is divided into two main phases:

- **Modeling phase**: **PREEvision**, one of the leading tools in the field of electrical and electronic (E/E) vehicle design, is used to create a complete logical model of the software architecture in accordance with the **AUTOSAR** standard. PREEvision enables the definition of components, services, interfaces, and signals, as well as the automatic generation of **ARXML files**, i.e., XML files in accordance with AUTOSAR standard that formally describe the elements of the system.

- **Implementation phase**: starting from the ARXML file produced in the previous phase, a **C++ application** is developed using the open source library **vsomeip**, based on the **SOME/IP** protocol, by BMW. This protocol, widely adopted in the automotive industry, is designed to support SOA architectures thanks to its lightness, interoperability, and service-oriented communication capabilities.

The result is a robust, scalable and coherent workflow that meets the current needs of the automotive industry, which is increasingly oriented towards the **Software-Defined Vehicle (SDV)** paradigm.

The thesis is structured as follows:

- **Chapter 2**: presents an overview of the state of the art, analyzing the main approaches and academic references concerning the evolution of automotive architectures towards service-oriented architectures. The context in which the proposed work is set is also discussed.

- **Chapter 3** introduces the fundamental concepts underlying SOA in the automotive sector. After comparing the monolithic and service-oriented approaches, the zonal architecture, V-Model and the role of Model-Based Systems Engineering (MBSE) as an advanced design methodology in the automotive world are explored in depth.

- **Chapter 4**: describes the tools used to support the design and implementation of SOA architectures in the automotive field, used in this thesis project. In particular, PREEvision is analyzed as a model-based tool, and the SOME/IP protocol is analyzed as middleware for communication between control units according to the service-oriented paradigm.

- **Chapter 5**: illustrates in detail the architectural modeling process, within PREEvision, created in this thesis project. All the main phases are addressed, from the definition of requirements to the communication level, including software and hardware design and mapping between levels.

- **Chapter 6**: describes the technical implementation of the modeled system, starting from the parsing of ARXML files to the generation of the JSON files needed to configure vSomeIP. The concrete implementation of the C++

applications for the provider and consumer roles is then explained, also tested on real embedded devices.

- **Chapter 7**: collects the final reflections on the work carried out, highlighting the results obtained, the original contributions, the limitations encountered, and the prospects for future development.

Chapter 2

State of the art

The automotive industry is undergoing a profound architectural transformation, in which software is no longer a simple support element but has become the main driver of innovation and value in vehicles. In this context, traditional architectures based on signal communications and tightly coupled control units are showing clear limitations in terms of the flexibility, modularity, and updating required by new-generation vehicles. With the number of ECUs ranging from a few dozen to several hundred depending on the specific system architecture, each communication is usually defined statically and rigidly during production. This configuration makes it extremely complex to introduce changes or new features, as it requires complete revalidation of the entire software system and intense coordination between the various teams involved, with development times that can last for months, significantly delaying innovation.

The answer to this complexity resides in the progressive adoption of development strategies based on modeling and modularity, in particular through the use of **Model-Based Design** (MBD) and the introduction of **Service-Oriented architecture**, now adopted as the foundation for the realization of **Software-Defined Vehicles** (SDVs). In this case, the vehicle is like a dynamic platform, updatable over-the-air and capable of integrating vehicle functions that are divided into independent **services**. This new approach allows for a decoupling of hardware and software.

The **Adaptive AUTOSAR** standard [16], an evolution of **Classic AUTOSAR**, was introduced to meet the new requirements of the automotive industry, offering greater architectural flexibility, service-oriented communication, and support for dynamic and updatable runtime systems. Contrary to Classic, which is based on static signal communication and deterministic cycles, Adaptive AUTOSAR enables interaction between components via methods, events, and fields, adopting modern protocols such as **SOME/IP** (Scalable service-Oriented Middleware over IP) and **DDS** (Data Distribution Service) and offering full support for Automotive Ethernet.

Despite these differences, Classic AUTOSAR still has a central role in hard real-time applications, such as sensor data acquisition and actuation. Its efficiency in terms of computational resource usage (CPU, memory) and simplicity in managing real-time use cases make it still suitable for low-complexity but highly critical scenarios.

In literature, numerous studies have analyzed this transition in the automotive context. Among these, *Sreeraj Arole*'s work [1] describes the migration of a legacy battery charging system from a monolithic architecture to a service-oriented architecture. The resulting system is built using a model-based software design approach, leveraging tools such as *Simulink*, *MATLAB System Composer*, and *C++ code autogeneration*.

The original system had strong couplings between software modules, making each update costly and subject to complete revalidation. The new architecture, on the other hand, redefined the system into independent services: for example, a *Sensor Extractor* service, services for charging protocols, and others, separated and orchestrated via SOME/IP events and methods.

The new system, tested in a Docker environment on Linux executables, demonstrated a 25% improvement in system response time and a 30% reduction in mean time to repair (MTTR). In addition to increased reliability (0.1% failures vs. 1% in the monolithic model).

These results demonstrate that the adoption of SOA, while involving a slight increase in CPU and memory usage, leads to substantial benefits in scalability, maintenance, and updatability of automotive software.

A second case of great industrial relevance is the study presented at the MECO 2020 conference [2]. This work addresses the implementation of SOA within a hybrid architecture, which combines traditional automotive networks (such as CAN and FlexRay) with a high-performance Ethernet backbone, on which a SOA platform is run. The goal of this balanced solution is to preserve the robustness and reliability of legacy networks for critical functions—such as brakes, steering, and transmission—while leveraging the flexibility and modularity of SOA for less safety-critical domains, such as infotainment and ADAS.

However, the introduction of SOA induces significant methodological challenges. In particular, the traditional V-Model used in the automotive industry is not well-suited to managing the dynamic, modular, and distributed nature of services typical of an SOA. This approach is too rigid to efficiently support the incremental validation of individual services and the management of dependencies between distributed components, requiring a fundamental new approach to the software development cycle.

In this context, the *SuperTuxKart* use case was presented, a video game integrated into the Mercedes CLA infotainment system. Although an external application, this game is able to communicate with the car's services, such as climate control, steering wheel, and pedals, demonstrating the interoperability possible thanks to SOA and SOME/IP middleware. The architecture also provides integration with *Update and Configuration Management* (UCM) modules, which enable secure over-the-air (SOTA) software updates, making the vehicle upgradeable like an IT system.

However, the study also highlights some challenges:

- the complexity of creating new services compatible with legacy networks;
- the security risk associated with exposing critical functions via APIs;
- the need to design strict governance of permissions and access to services from the outset.

Another significant contribution to the transition to service-oriented architectures is the study by Obergfell, Kugele, and Sax [3], which also proposes a model-based design approach for the synthesis and analysis of SOA architectures in the automotive sector. The work describes a modeling framework consisting of three main components: a *meta-model* for the formalization of services (including events, methods, and temporal and functional attributes), a *Design Space Exploration* (DSE) engine for the optimized allocation of services on hardware resources, and an artifact generator for *run-time* verification of temporal properties, with support for dynamic reconfiguration.

The method has been validated through a case study on an environmental perception function for automated driving, demonstrating how optimal service deployment can be achieved by taking into account multiple constraints, including computational resource usage, functional safety levels *(ASIL)*, and execution latencies.
This approach shows itself to be particularly effective in view of the new requirements introduced by advanced autonomous vehicles according to the *SAE J3016* classification. Starting from level 3, in fact, the E/E architecture of vehicles must be able to dynamically support complex services, ensuring flexibility, safety, and continuous updatability.

This work is part of this line of research and development, proposing a concrete use case that demonstrates the effectiveness of designing and implementing a SOME/IP service on embedded control units, starting directly from modeling in **PREEvision**. Compared to previous work, the main contribution of this thesis lies in the complete closure of the **Model-Based Systems Engineering** (MBSE) cycle with the creation of two **autonomous C++ applications** — provider and consumer — that really interface with each other as modeled and the real SOME/IP communication test was also performed between embedded control units and not only virtually. Furthermore, it has been shown how it is possible to test communication between control units, bypassing OEMs, Tier 1s, and other external suppliers, significantly speeding up internal benchmarking.

Another innovative aspect concerns the introduction of **custom attributes** in service modeling. These elements, already present in the design phase, are designed to support, in the future, the development of a system for dynamically moving services between control units in the case of overload or malfunction, with a view to dynamic reconfiguration and greater system resilience.

The work therefore is based upon previous studies, but represents an extension of their application, which may inspire future activities in both academia and industry.

Chapter 3

Automotive Background

This chapter analyzes the architectural evolution of vehicles, starting from **traditional monolithic software** solutions to the more recent adoption of **Service-Oriented architecture** and **zonal hardware architecture**, which are essential for enabling the transition to **SDVs**.

It also examines the **V-Model development** process and compares it with its more advanced evolution, the **Model-Based Systems Engineering** approach.

3.1 SOA vs Software-Oriented architecture

Traditionally, vehicles have been built using a **software-oriented approach** but with increasing system complexity and the demand for greater flexibility, scalability, and updatability, the industry is moving toward **Service-Oriented architecture**.

3.1.1 Software-Oriented architecture: a monolithic approach

In a classic software-oriented system, vehicle functions are implemented as **monolithic** software modules, tightly integrated with specific ECUs. Each ECU is responsible for a well-defined function, and the entire software is **strongly dependent** on the hardware on which it runs. This approach, while initially successful for less complex systems, has several limitations:

- **Difficult to update**: every change requires re-flashing the entire system.
- **Low reusability**: the code is written for a specific hardware configuration, and the same function must be implemented multiple times if requested in a different context.
- **Poor scalability**: adding new features often requires a complete revision of the system.
- **High complexity of testing and integration**: every change requires end-to-end testing of the entire system.

Software development for the automotive industry is also characterized by a complex division of work between the OEM and a wide ecosystem of suppliers, each providing control units that implement their own software or firmware. Integrating all the hardware and software to compose the final vehicle requires a **delicate** and **highly coordinated process**. The absence of modularity makes **any change a costly and time-consuming procedure**, significantly increasing time-to-market.

With the increase in digital features in vehicles, software complexity has also grown exponentially. In 2010, the average car contained around 10 million lines of code; by 2016, this had exceeded 150 million. This growth has had a direct impact on costs: in 2018 alone, software problems caused recalls for millions of vehicles, with economic damage of more than 17 billion [5].

So, the entire industry is undergoing a historic transition: from hardware-defined vehicles to software-defined vehciles.

However, the growing centrality of software brings new challenges. Industry stakeholders now consider aspects such as **software quality**, **maintainability**, and **safety** to be fundamental requirements for ensuring proper vehicle performance, especially in view of autonomous and safety-critical features. Furthermore, software maintenance costs now exceed 67% of the total lifecycle. Even a simple change can take weeks or months to validate, increasing operating costs and delaying the release of updates [5].

In summary, monolithic architectures have reached the limits of their scalability and efficiency. The evolution of the automobile requires a new software development model that is more flexible, modular, and capable of addressing increasing complexity. This model is represented by SOA, which will be analyzed in the following section.

3.1.2 Service-Oriented architecture: modularity and flexibility

SOA is the answer to the need for **modularity** and **reusability** in automotive software. In an SOA, vehicle functions are implemented as independent **services**, published and dynamically discovered through an intermediate layer called **middleware**, which is located between the operating system and the applications and uses **standardized interfaces** (APIs). This level of abstraction allows the software to be **decoupled** from the underlying hardware, allowing services to be transferred from one ECU to another or reused on different platforms, regardless of the specific vehicle configuration. This is also shown in Figure 3.1.

Adopting a secure SOA framework simplifies all phases of development, from design to maintenance, enabling a more agile and efficient software lifecycle.

Each service, such as speed control or adaptive lighting management, can be developed, tested, and updated **independently**. Communication between services takes place according to standardized paradigms such as **RPC** (remote procedure calls) or **publish/subscribe**, often implemented via protocols such as **SOME/IP** or **DDS**. This allows for a distributed architecture, which can also be extended to third parties, where updating a single service does not require reconfiguring the entire system [7].

In the automotive context, SOA is typically implemented following the **AUTOSAR Adaptive** standards, which work alongside **AUTOSAR Classic**, designed for real-time and deterministic components. The Adaptive environment is designed for data-intensive systems and allows the use of technologies such as *POSIX*, *C++*, and *middleware* protocols such as SOME/IP, mentioned above and now among the most widely used in the industry for ECU interoperability.

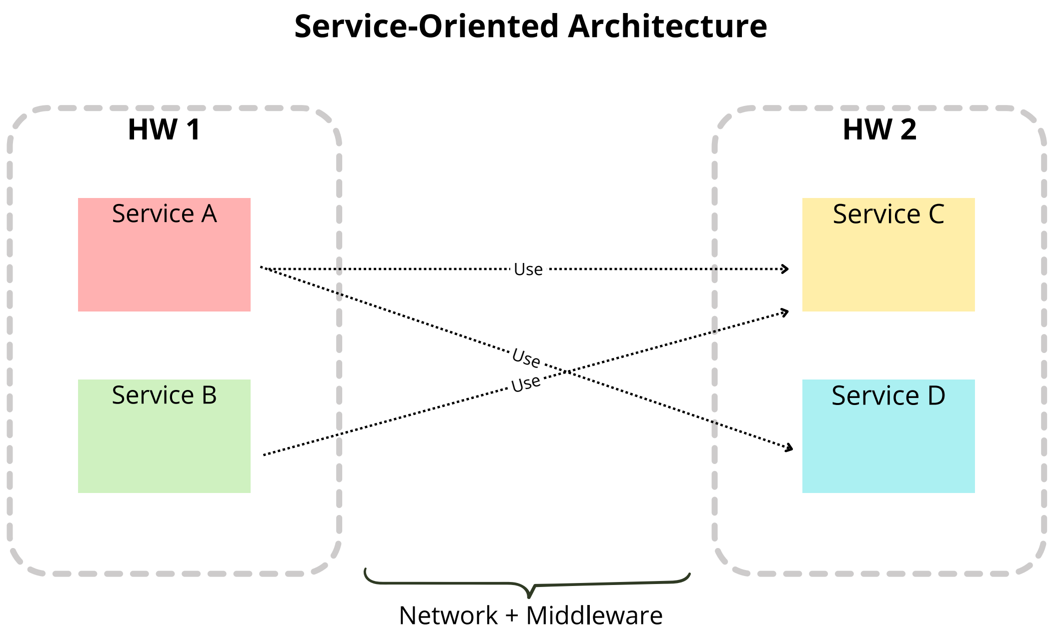


Figure 3.1: Service-Oriented architecture

SOA features The key features of an SOA include [6]:

- **Loose coupling**: services are independent and communicate through standard middleware (such as SOME/IP).
- **Modularity**: individual services can be developed, tested, and deployed separately.
- **Portability**: services can be run on different ECUs or moved during the vehicle's life cycle.
- **Scalability and reusability**: a service can be reused in multiple vehicles and configurations, promoting development efficiency.

- **Dynamic updatability**: even after the vehicle is purchased, services can be updated or enhanced with new features.
- **Abstraction and autonomy**: each service is defined only by its interfaces and manages its own lifecycle autonomously.
- **Composability and discoverability**: services can be composed into complex applications and are dynamically discoverable, even by third parties.

This approach drastically **reduces integration complexity and enables faster development cycles** with fewer errors and fewer testers. According to industry estimates, validation cycles can be reduced from several months to a few weeks, with positive impacts on time-to-market and operating costs.

SOA also **redefines collaboration between OEMs and suppliers**: while the OEM keeps responsibility for the general architecture and reference framework, the implementation of individual services can be outsourced to external suppliers or third-party developers. This open model allows new players — even outside the traditional automotive ecosystem — to develop applications for the vehicle without needing detailed knowledge of the underlying hardware. This encourages competition, expands the available functionality, responds quickly to market needs, and brings the automotive user experience closer to that of personal devices such as smartphones or smart TVs.

3.1.3 Comparing traditional and SOA approaches

As discussed in sections 3.1.1 and 3.1.2, vehicle software architecture has been evolving from the traditional monolithic model towards a service-oriented paradigm.

Monolithic architectures concentrate all functionality in a **single integrated system**, with a **strong dependency** between components. This approach, historically adopted in the automotive industry, offers simplicity in the initial development phases but is rigid and not very flexible when it comes to maintenance, updates, or scalability.

Service-oriented architecture, and more generally a **microservices** model that shares modularity and functional independence with SOA, introduces a more modern paradigm. The software is divided into **independent** and **loosely coupled** components, each with a well-defined functional logic and the capacity to operate autonomously. This structure allows for more agile development, dynamic updates, and greater adaptability to market changes or the needs of the end user.

Below is a direct comparison of the two models, highlighting their advantages, disadvantages, and practical impacts in the field of automotive software development.

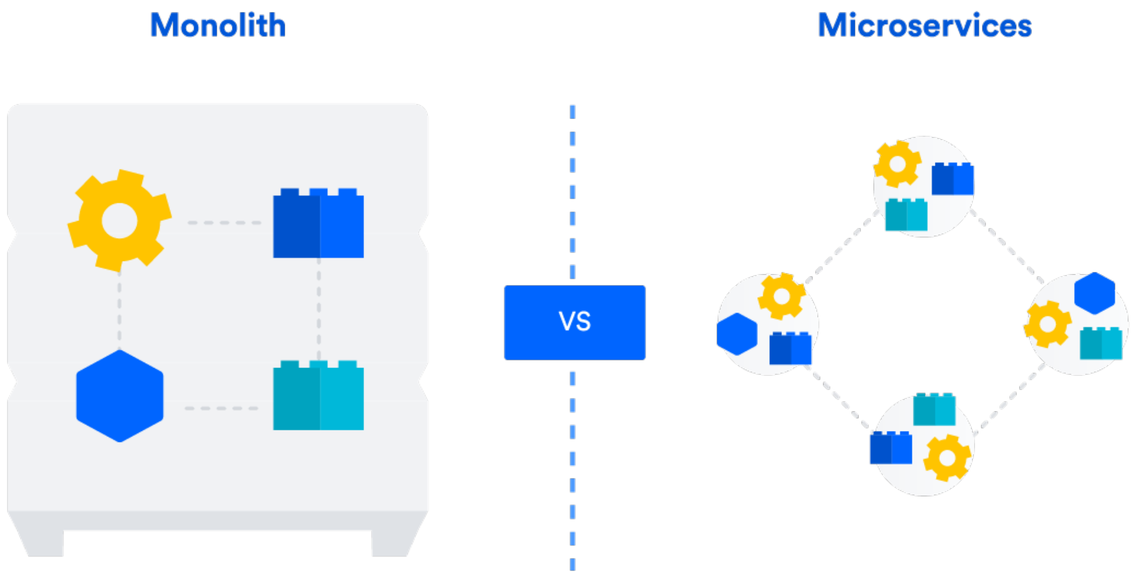


Figure 3.2: Monolith vs microservices [22]

Time-to-market and flexibility The reuse of software components in the shape of services makes the development of new features much faster, as developers do not have to build every feature from scratch. In a monolithic architecture, every new feature may require integration into an already complex and interdependent system, significantly slowing down release times. In contrast, an SOA-based system allows for frequent iterations and shorter development cycles.

Extensibility and scalability SOA architecture makes it easier to adapt software to new markets and platforms. Each service can be adapted or replicated in different contexts without having to redesign the entire application. This is particularly useful in the automotive industry, where different hardware configurations can coexist. Furthermore, thanks to the distributed nature of SOA, it is possible to selectively scale the most requested individual services, whereas in a monolithic system the entire application must be scaled, with a high waste of resources.

Maintenance and upgradeability In a monolithic system, even the smallest change requires global retesting and redistribution of the entire application, increasing costs and risks. SOA allows for timely and localized updates, improving software lifecycle management and drastically reducing the risk of bug propagation.

Debugging Debugging monolithic applications is more direct as it takes place in a single environment, but can become extremely complex in large systems. In microservices, each component must be analyzed separately, but their independence allows for better traceability of problems.

Implementation and deployment Monolithic applications are easier to deploy, as they require the installation of a single software package. However, this approach limits flexibility and introduces infrastructure constraints. Microservices require more sophisticated deployment management (e.g., containers, orchestrators), but offer independent deployment, which is essential for continuous delivery scenarios.

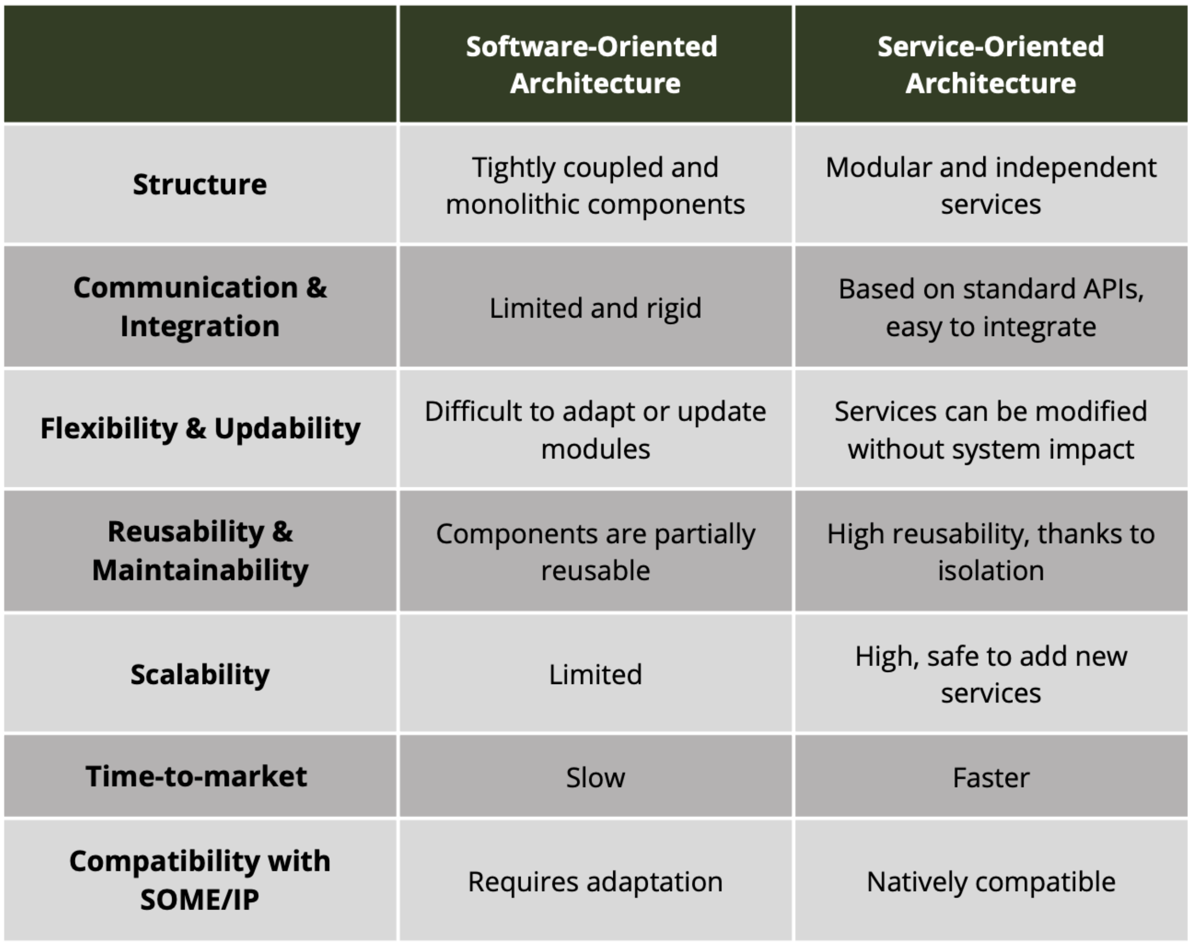
Efficiency and costs Although the initial cost of a monolithic project may seem cheaper, microservices are more cost-effective in the long run. They allow for more efficient use of computational resources (horizontal scalability) and reduce maintenance costs. In particular, independence from a specific hardware platform avoids the need for costly adaptations to legacy systems.

Reliability and resilience The functional separation of services in an SOA architecture allows for greater resilience: a localized malfunction has minimal impact on the system as a whole, reducing the risk of critical downtime. In a monolithic structure, on the other hand, a bug can compromise the entire application, with serious consequences for the performance of the vehicle. [8]

An overview of the advantages and disadvantages of the two architectures summarized is shown in the Figure 3.3.

In summary, the comparison between the two architectures confirms the findings already outlined in the theoretical analysis of Service-Oriented Architecture (Section 3.1.2): the SOA approach offers a concrete and structured solution to the increasing complexity of automotive software systems. Its adoption entails not only a technological shift, but also a cultural and organizational one.

While monolithic solutions may still be suitable for simple or legacy systems, the growing demand for dynamic and flexible E/E (Electrical/Electronic) architectures clearly points to service-oriented paradigms as the foundation for future automotive developments, particularly in enabling advanced scenarios such as **Vehicle-to-Everything (V2X)** communication, **cloud-based integration**, and **dynamic service orchestration** in connected vehicles.



	Software-Oriented Architecture	Service-Oriented Architecture
Structure	Tightly coupled and monolithic components	Modular and independent services
Communication & Integration	Limited and rigid	Based on standard APIs, easy to integrate
Flexibility & Updability	Difficult to adapt or update modules	Services can be modified without system impact
Reusability & Maintainability	Components are partially reusable	High reusability, thanks to isolation
Scalability	Limited	High, safe to add new services
Time-to-market	Slow	Faster
Compatibility with SOME/IP	Requires adaptation	Natively compatible

Figure 3.3: Comparison between Software-Oriented and Service-Oriented architectures

3.2 Zonal architecture: evolution of E/E

As a natural extension of the SOA described in the previous section, vehicle hardware architecture is also undergoing a transformation. In response to increasing functional complexity, the centrality of software, and a need to reduce costs, classic decentralized E/E architectures based on a multitude of dedicated ECUs and extensive wiring are giving way to **zonal architecture**: a more modular and scalable structure that integrates well with SOA principles.

3.2.1 Overcoming the decentralized model

Traditionally, automotive E/E architectures are based on a decentralized model: each function is implemented in a dedicated ECU connected to a network of wiring that extends through the entire vehicle. This structure has supported the evolution of vehicles for years, but has reached its limits in terms of weight, cost, maintainability, and scalability. Wiring, for example, is now the third heaviest and most expensive component in a modern car, after the engine and battery [9].

With the continuous introduction of new features — ADAS, infotainment, connectivity — the continuous addition of ECUs and cables has further increased complexity. The hardware-first paradigm, where software is adapted to existing hardware, has proven inefficient. In contrast, modern architectures follow a software-first logic, where software defines the architecture and hardware requirements, creating what we now call SDVs, as mentioned in previous sections.

The zone concept Zonal architecture divides the vehicle into macro areas (zones), each managed by a high-performance zonal ECU (often a multicore SoC). These ECUs are connected to local sensors and actuators and communicate with each other via a high-speed **Ethernet backbone**. Zones are defined based on physical location (e.g., front left zone) or function (e.g., infotainment, active safety). This enables:

- a drastic reduction in the number of ECUs,
- significant optimization of wiring,
- more modular and scalable management of functionality,
- easier maintainability,
- hardware consolidation that allows multiple functions to be unified into multifunctional control units.

Features that in the past would have required a dedicated control unit, such as adaptive lighting or window management, can now be performed by zonal ECUs or even by a central High Performance Computer (HPC), which acts as a coordinating node for the entire vehicle.

Ethernet as a communication backbone The heart of the zonal architecture is the Ethernet backbone, which enables data exchange between zones and to any central HPCs. Speeds reach 1–10 Gbps, sufficient to handle video, radar, LIDAR, and other data-intensive applications. The use of Ethernet, together with support for Time-Sensitive Networking (TSN), ensures minimal latencies and determinism, which are essential for active safety systems.

Furthermore, a multi-protocol approach is still necessary: peripheral local networks can continue to rely on traditional protocols such as CAN, LIN, or FlexRay, but interfacing transparently with the Ethernet backbone via zone controllers.

One of the main advantages of this model is **flexibility**: software services can be dynamically reassigned to different ECUs or HPCs, depending on operating conditions or available resources. The hardware thus becomes programmable, and the functional configuration can be varied not only during production but also after sales, with OTA (Over-The-Air) updates.

In economic terms, it is estimated that the transition to a zonal architecture can significantly reduce development, maintenance, and production costs. The simplification of wiring, standardization of modules, and reduction of dedicated ECUs lead to a drastic reduction in **vehicle weight**, also contributing to greater energy efficiency, which is particularly important in electric vehicles.

Finally, thanks to the close integration between zones and the fast communication guaranteed by the Ethernet backbone, the vehicle system can respond in a coordinated way to critical events. In scenarios such as emergency braking, sensors in the front zone (cameras, radar, lidar) send signals in real time to the zonal ECU or the HPC. These, via the network, activate a series of events in a matter of milliseconds: brakes are applied, seat belts are tightened, headlights flash, windows are lowered, airbags are activated, and warnings are displayed on the screens. All this is achieved by exploiting the synergy between local sensors and central logic [10].

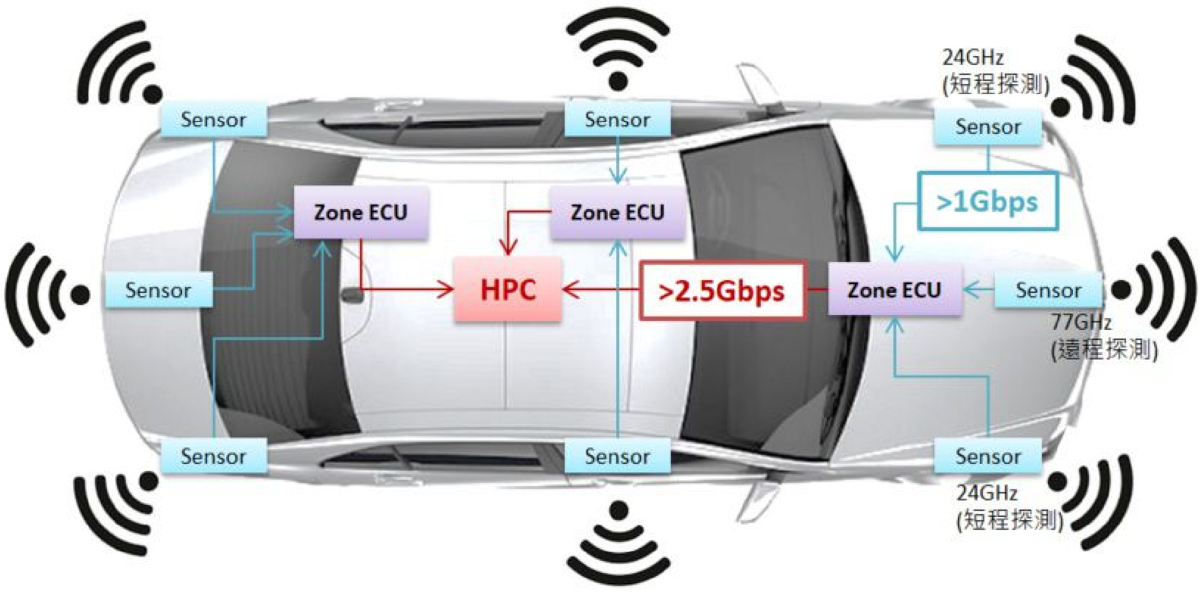


Figure 3.4: Zonal topology [20]

As we will see in the following chapters, this hardware architecture, combined with service-oriented middleware, provides the technical and conceptual basis for the development of modern, agile, and interoperable SDV platforms.

3.3 The V-Model

One of the most widely adopted development models in the automotive industry is the so-called **V-Model** (where V stays for verification and validation), an approach that divides the development process into two distinct but mirrored branches: requirements gathering, design and software development on the left branch, while validation are on the right branch, followed by release. Each definition activity on the left side has a corresponding validation phase on the right side, which allows for traceability and quality control of the product.

Each of these phases begins with a formal document and ends with another, allowing different parties to perform each phase with minimal overlap.

In the automotive context, the V-Model is the basis for fundamental regulatory standards such as **ISO 26262**, which defines requirements for functional safety in the automotive industry, and **ASPICE** (Automotive Software Process Improvement and Capability dEtermination), which allows OEMs to check the quality of the software supplied to them. The widespread adoption of these standards has further established the V-Model as the dominant development framework for safety-critical systems, including automatic braking, Advanced Driver Assistance Systems (ADAS), and autonomous driving technologies [12].

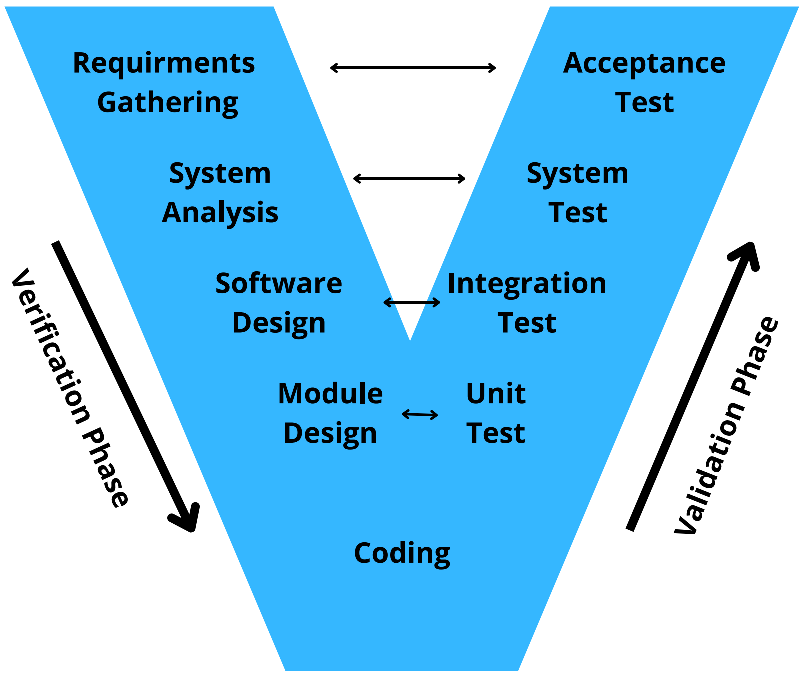


Figure 3.5: V-Model

The process begins with **requirements gathering**, in which the desired functionalities and operating modes of the system are precisely formalized. It is essential that this phase is conducted accurately, as the entire success of the development depends on the clarity and consistency of the initial requirements.

Next, **system design** defines the overall architecture, the software and hardware components, the ECUs involved, their interaction, and the network topology. The design can make use of advanced modeling environments (such as MATLAB/Simulink or PREEvision) that allow the behavior of the system to be simulated in realistic scenarios. The first tests carried out on these models are called **Model-in-the-Loop** (MiL).

This is followed by **software development**, often implemented in C language and following AUTOSAR guidelines. During this phase, automatic tests **Software-in-the-Loop** (SiL) are often performed to validate the behavior of the software in simulated environments, still without physical hardware.

Finally, the **integration** phase gets started, in which the software components are assembled and implemented on the physical ECUs. All these phases are formally documented and tracked. Testing in this phase includes both simulations in **Hardware-in-the-Loop** (HiL) environments and tests on real vehicles. [11]

From Figure 3.5, we can see the different phases. On the right-hand side, we have the specific tests associated with each phase: **unit testing** for individual software modules, **integration testing** for interactions between modules, **system testing** to verify the entire system, and **acceptance testing** to ensure compliance with the initial requirements. Techniques such as **MiL**, **SiL**, and **HiL** are used to simulate real-world scenarios and validate system behavior.

The main advantages of the V-Model are:

- **Structured and transparent approach**, with clearly defined phases. It synchronizes the development, testing, and validation phases.
- **Advance test planning**, which allows bugs to be identified early, saving substantial costs.
- **Regulatory compliance**, thanks to the production of accurate documentation and the traceability of design decisions, it is possible to demonstrate compliance with automotive safety standards.
- **High product quality**, facilitated by continuous verification and formal validation.
- **Risk minimization**, thanks to early testing.

However, this model also has some limitations, especially in the view of the new demands of the automotive industry:

- **Rigidity**: it is not very flexible in handling late changes or iterative cycles, making customer feedback complicated;
- **High costs and long lead times**, especially for prototyping or limited production projects;
- **Long development cycles**, which are incompatible with the need for rapid time-to-market;
- **Laborious documentation**, which can increase overall costs in terms of time, resources, and expenses;

- **Difficulty of application in agile contexts**, where requirements evolve dynamically.

For this reason, more dynamic approaches are emerging that integrate agile methodologies into the traditional V-Model (Agile-V), or that reinterpret its structure in a more iterative and collaborative way, such as the **DevOps** paradigm or continuous verification and validation throughout the software development lifecycle, integrating the phases rather than separating them.

One of the most promising developments in this sense is the adoption of **Model-Based Development**, a methodology that uses models to represent software behavior. It allows testing and simulation to be performed as early as the design phase using tools such as MATLAB/Simulink, increasing the efficiency of the V-model. This approach has made it possible to detect bugs earlier, achieve greater safety and reliability, and reduce development time in the early stages. The model, validated through MiL testing, then becomes the basis for the embedded code, which is either generated automatically or written by hand.

Despite its limitations, the V-Model remains an essential reference for ensuring **safety** and **quality** in automotive software development. However, as will be seen in the next section, the increasing level of complexity of systems and the expansion of SDVs require an evolution of design methodologies.
Although many of the phases followed still reflect a structure inspired by the V-Model — such as requirements definition, architectural design, and subsequent functional validation — the approach adopted has been deeply model-based and iterative.
In this context, **Model-Based Systems Engineering MBSE** represents a natural extension and evolution of the V-Model, and the project of this thesis is a concrete example of the evolution of the traditional paradigm.

The choice of a model-based approach, also supported by the automatic generation of artifacts (such as ARXML files or SOME/IP configurations), has allowed for greater traceability, simulability, and modularity compared to a workflow based exclusively on static documentation and post-development verification. This made it possible to iterate quickly between design, simulation, and validation, anticipating errors and improving the efficiency of the development cycle, even on embedded platforms.

3.4 Model-Based Systems Engineering in the automotive sector

MBSE does not aim to replace the V-Model entirely, but rather evolves it by introducing an iterative and incremental process centered on the use of executable **digital models**. Unlike the traditional V-Model, where testing and validation occur mainly in the final stages, MBSE enables early simulation of system behavior, allowing for the prompt identification of errors, inconsistencies in requirements, and integration issues between software and hardware components. This leads to **cost reduction and faster time-to-market**.

This model-driven approach responds to the increasing need for **modularity**, **traceability of requirements**, and **early validation**, which traditional methods based on manual documentation, reliant on static diagrams and isolated specifications, struggle to address. In MBSE, the hardware, software, and communication architectures are represented through formal, executable models that describe system behavior, structure, and interactions in a consistent and automated way. These models are used throughout the development lifecycle to design, simulate, verify, and document the system, and sometimes even to generate code.

The advantages of MBSE extend to all levels of the development cycle, particularly in highly complex sectors such as aerospace, medical, and, of course, automotive.

In the automotive industry, this methodology has gained wide adoption, supported by tools such as **PREEvision**, **AUTOSAR**, and **MATLAB/Simulink**. In particular, PREEvision — used in this thesis project — enables the integration of requirements, logical architecture, and network configuration into a unified platform, supporting a consistent and realistic implementation of architectures.

Furthemore, MBSE in this context, is better suited to handling changes in requirements during development, as all stakeholders (OEMs, Tier-1s, and validators) work on a shared representation of the system.

MBSE integrates effectively with advanced engineering practices, including:

- **Data centralization**: models consolidate requirements, architecture, logic, and tests into a single platform, reducing ambiguity and inconsistencies.
- **Architectural design**: logical, software, and hardware structures and their interactions can be described graphically.
- **Early simulation and validation**: models enable scenario simulation and behavior validation, identifying design errors before the implementation phase.
- **Life cycle management**: all system artifacts are traceable; requirements are mapped to software and hardware components, improving development management.
- **Integration with advanced toolchains**: many MBSE tools support export/import in standard formats such as ARXML (AUTOSAR), facilitating integration into the industrial workflow.

The model-based approach can be divided into several branches, each with specific focuses and outputs:

- **Model-Based Development (MBD)**: focused on the development of algorithms and control logic; produces C code autogenerated from environments such as MATLAB/Simulink, ready for embedded integration.
- **Model-Based Systems Engineering (MBSE)**: focused on modeling software/hardware architectures and functional system requirements; uses tools such as PREEvision or Capella.
- **Model-Based Testing (MBT)**: enables the automatic generation of test scenarios (MIL, SIL, HIL) from models; mainly used in the validation phase.
- **Model-Based Communication Design**: focuses on the design and configuration of communication networks, generating standard configuration files (DBC, ARXML) for protocols such as CAN, FlexRay, or SOME/IP.

A typical use case in the automotive sector could follow the flow described below:

- Modeling of functional and safety requirements;
- Definition of the software architecture (e.g., AUTOSAR Adaptive components) and hardware (e.g., zonal ECU);
- Validation of communication flows and network configurations (e.g., radar $\rightarrow$ control unit $\rightarrow$ actuators);
- Automatic generation of ARXML configuration files.

MBSE, therefore, provides a solid methodological basis for the development of **distributed**, **interoperable**, and **scalable** architectures, such as service-oriented and zone-based, which will be the subject of analysis in the following chapters.

Starting from the next chapters, **PREEvision** modeling will be introduced, the tool adopted for architectural modeling in the project described in this thesis, showing how it applies the principles illustrated in this chapter.

Chapter 4

Tools and Technologies for SOA Automotive Development

This chapter analyzes in detail the tools and protocols used for designing service-oriented architectures in the automotive sector. In particular, it explores the use of PREEvision for E/E modeling and the SOME/IP protocol for implementing communication between ECUs.

4.1 PREEvision: a model-based tool for E/E design

In the context of the growing complexity of electrical and electronic (E/E) systems in modern vehicles, advanced architectural modeling tools are playing an increasingly central role. Among these, **PREEvision**, developed by *Vector Informatik* [1] , is one of the most advanced MBSE tools and facilitates the design and management of complex E/E architectures. It supports the entire design lifecycle, from requirements collection to wiring generation, integrating software, hardware, communication modeling and validation in a consistent and shared environment.

PREEvision, follows a model-based approach, which allows all dimensions of a vehicle architecture to be represented in a single, centralized platform, as mentioned in the previous chapter. Following the basic principles of systems engineering — *abstraction*, *decomposition* and *reuse* — the tool allows complex projects to

[1]Vector Informatik is the leading manufacturer of software tools and integrated components for the development of electronic systems and their interconnection with many different systems, from CAN to automotive Ethernet.
Vector has been a partner to automotive manufacturers and suppliers and related industries since 1988.

be managed through a hierarchy of interconnected levels, each of which represents a specific view of the system. (Figure 4.1) This makes it possible to model different product lines and architectural variants [15].

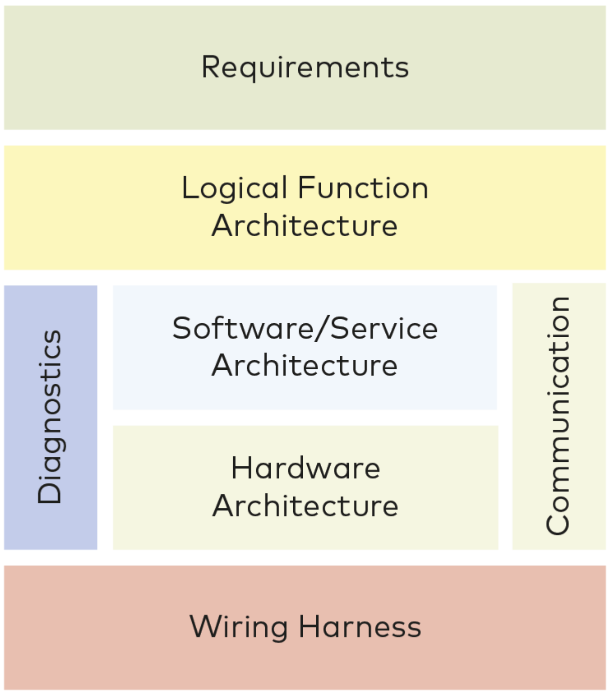


Figure 4.1: PREEvision layers [15]

PREEvision's integrated data model includes:

- **Requirements and change management**: representation of functional and non-functional requirements and enables complete tracking of changes and dependencies between system layers.
- **Logical architecture**: modeling of functions in abstract terms independent of implementation.
- **Software architecture**: definition of software components, services, interfaces according to AUTOSAR standard.
- **Hardware architecture**: modeling of ECUs, sensors, actuators, and other physical devices.
- **Communication design**: facilitates the configuration and optimization of CAN, LIN, Ethernet, and FlexRay networks.
- **Wiring and geometry**: design of the wiring harness and physical routes in the vehicle.
- **Simulation and validation**: includes tools for consistency analysis and system behavior simulation, even before physical implementation.

These layers are organized vertically, representing a flow of increasing abstraction, from geometry to software. In parallel, the horizontal direction allows functional decomposition (both top-down and bottom-up), while a third orthogonal dimension allows the modeling of variants and reuse on each layer of the model [14].

The **connection between the different layers** of the model is ensured through a system of **mappings**, which logically links requirements, functions, software components, ECUs, and wiring, allowing for consistent management of dependencies and changes at every stage of the design.

Every element modeled in PREEvision is an integrated part of a **centralized database**, accessible by multiple users and locations. This ensures not only **end-to-end traceability** of every edit, but also effective collaboration between OEMs, suppliers, and multidisciplinary teams, ensuring consistency even when changes or adaptations are made for new vehicle generations.

A further strength of the multi-level approach adopted by PREEvision lies in the **separation between logical architecture and physical implementation**: many vehicle functions can be modeled at an abstract level, independent of hardware and software, allowing the **logical structure to remain stable over time**, even when the underlying platforms (software and hardware) are updated.

4.1.1 Support for AUTOSAR classic and adaptive

One of the distinctive features of PREEvision is its native support for AUTOSAR standards [16], both in the Classic and Adaptive versions. For the latter, which has been the main focus in this thesis project and is intended for software-defined vehicles and service-oriented communications, PREEvision provides a dedicated user interface for defining:

- Services and interfaces (SOME/IP, DDS)
- Application, service, and ECU manifests
- Ethernet topology and software-to-hardware mapping

The *Adaptive Explorer* enables modeling in line with AUTOSAR Adaptive standards, guiding the user through the design of services, software, and network topologies. In addition, it enables the automatic export of all relevant configurations in ARXML format — including service interface descriptions, application and service instance manifests — thus enabling integration with external toolchains for code generation, system configuration, and behavior simulation, improving interoperability with other tools and project partners.

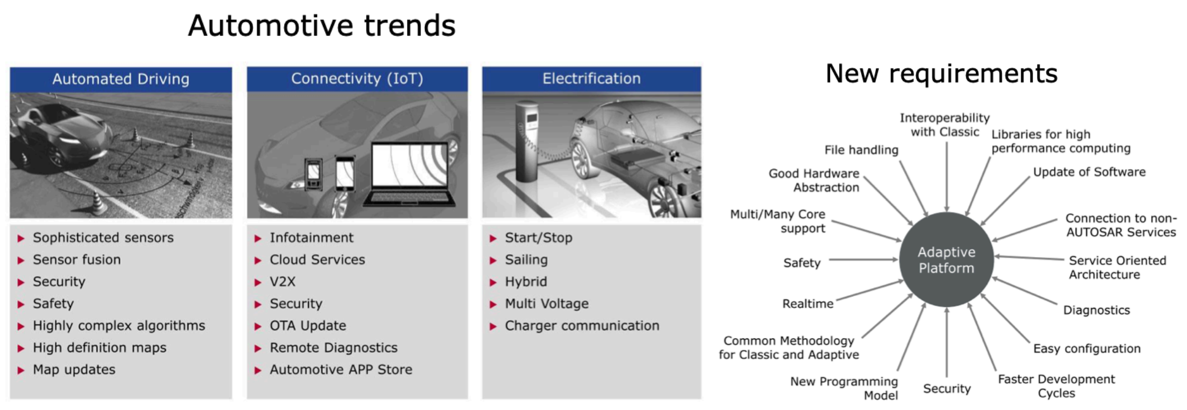


Figure 4.2: AUTOSAR adaptive [21]

4.1.2 Advantages in the automotive sector

The use of PREEvision in the automotive sector offers numerous advantages:

- **Reduction in development time and costs**, thanks to early validation and automation.
- **High design quality**, facilitated by simulations, consistency analyses, and compliance with standards.
- **Multi-site collaboration** and centralized change management.
- **Flexibility and reuse** of models across multiple vehicle variants and generations.
- **Scalability**, from small projects to complex SDV architectures.

In the project described in this thesis, PREEvision was used for service-oriented architecture modeling.

It allowed to define software components and design SOME/IP interfaces between ECUs, define functional requirements, keeping consistency between the different logical and physical levels. This approach has made it possible to transform the principles of MBSE into concrete artifacts, significantly reducing development times and improving the consistency of the entire design process.

4.2 SOME/IP protocol: middleware for Service-Oriented architectures

In the context of the evolution of vehicles towards SDVs, the need to manage an increasing amount of data in real time and in a flexible way has driven the automotive industry to move away from traditional signal-based communication models (such as CAN, LIN, and FlexRay) towards a service-based paradigm. This change has made it necessary to adopt technologies that guarantee **high performance**, **interoperability**, and **scalability**. (Figure 4.3)

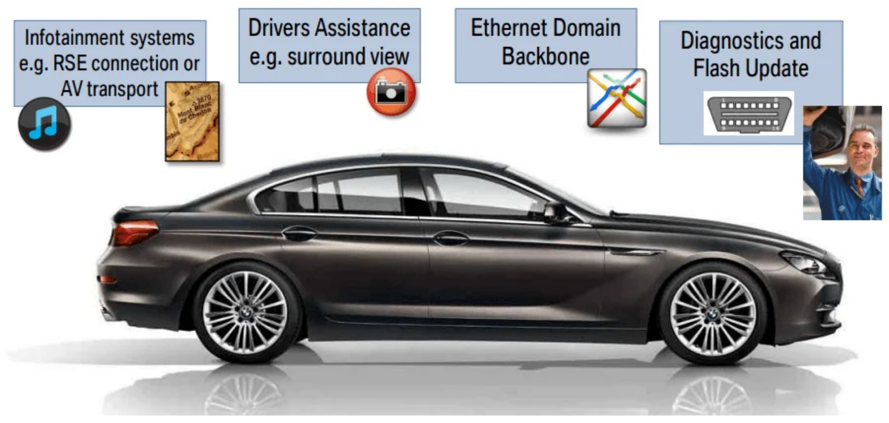

Infotainment systems e.g. RSE connection or AV transport
Drivers Assistance e.g. surround view
Ethernet Domain Backbone
Diagnostics and Flash Update

Figure 4.3: Typical use cases of the SOME/IP protocol in automotive systems [18]

In this scenario, Ethernet has emerged as the reference standard for meeting new communication requirements inside and outside the vehicle, thanks to its reliability, bandwidth (up to 1000 Mbps), and compatibility with established protocols such as TCP/IP and UDP. However, the direct use of Ethernet is not sufficient to meet the specific requirements of the automotive sector. It is therefore necessary to introduce middleware that enables fundamental features such as:

- *Data serialization* — ensures interoperability between ECUs with different operating systems and hardware architectures by transforming data into and from on-wire representations appropriate for network transmission.
- *Remote Procedure Call (RPC)* — allows remote invocation of functions between client and server ECUs, with or without return values, i.e., the client can request data as a response or simply call a function to execute certain tasks on the server side.

- *Service Discovery (SD)* — allows dynamic discovery of services available on the network. In a service-oriented architecture, it is essential that a service can be found.
- *Publish/Subscribe* — enables efficient transmission of data only to clients that explicitly request it.

The middleware that implements all these features is **SOME/IP**, introduced by BMW in 2011 and now integrated into the **AUTOSAR Adaptive** standard. The protocol is the best way to set up dynamic communication between ECUs in a service-oriented architecture.

The behavior is also illustrated in the Figure 4.4.

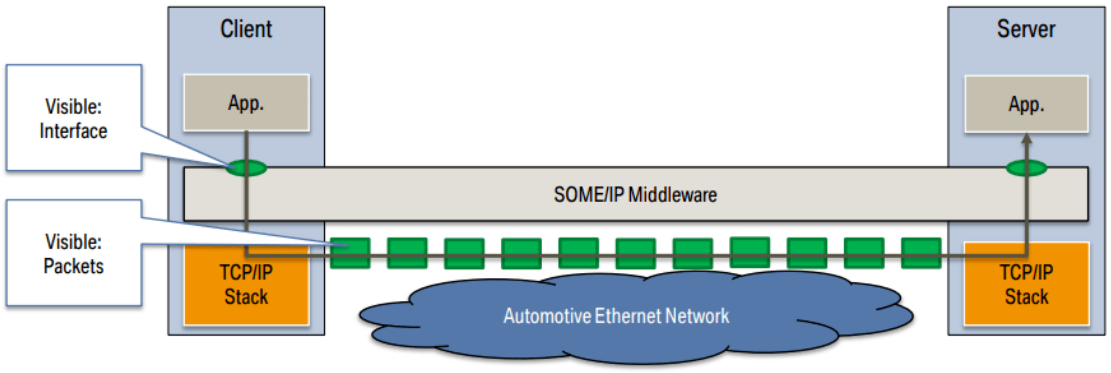


Figure 4.4: How SOME/IP works on Ethernet backbone [18]

From signal-based model to service-oriented communication In traditional architectures, communication between ECUs was statically defined: each signal was transmitted regardless of the actual need of the receivers. This approach, suitable for systems with static software and strongly linked to hardware, tends to overload the network with redundant data, limiting flexibility and the possibility of updating. However, assuming that the software would never change, the signal-based model was the most suitable solution.

With the SOA paradigm, data transmission only occurs upon explicit request. Server ECUs must be aware of which clients are interested in the data. This mechanism allows the software to be decoupled from the hardware, improving modularity, scalability, and efficiency.

Client-Server model SOME/IP follows a **client-server architecture**: a provider offers services, and one or more clients can consume them.

Every service can expose [18]:

- **Methods**, which can be invoked via RPC calls;

- **Events**, which notify changes asynchronously;
- **Field**, data structures that can be updated with getters, setters, and notifiers method.

Supported communication types include:

- **Request/Response** — The client receives a direct response to the request from the server.

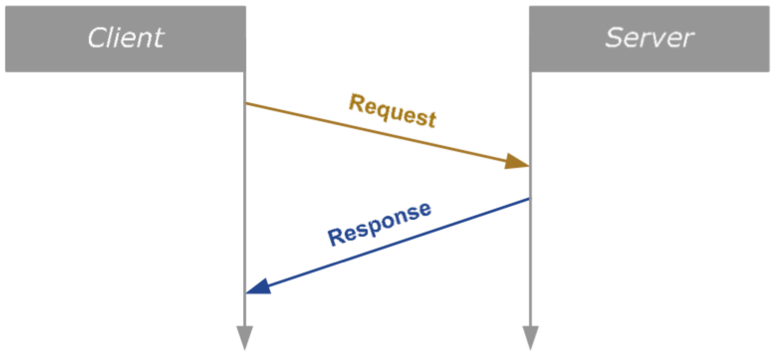


Figure 4.5: Request/Response

- **Fire-and-Forget** — The client sends a request without expecting a response.

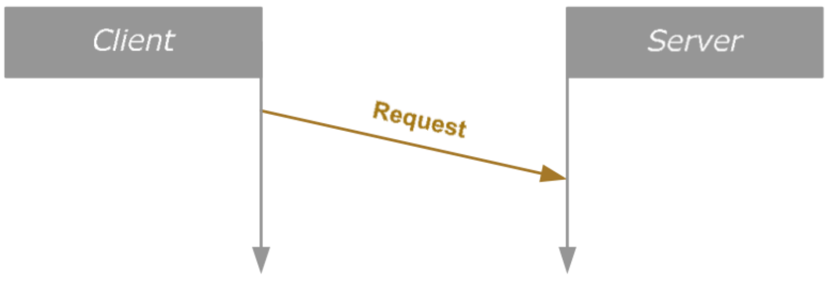


Figure 4.6: Fire-and-Forget

- **Event** — The server sends asynchronous notifications to subscribed clients, either cyclically or when there is a change in one of the attributes managed by the server.

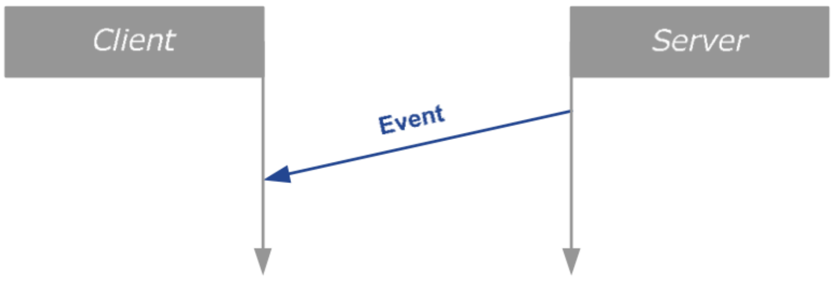


Figure 4.7: Event

- **Field** — Optionally includes getter (to read the value), setter (to update the value), and notifier for dynamic data access.

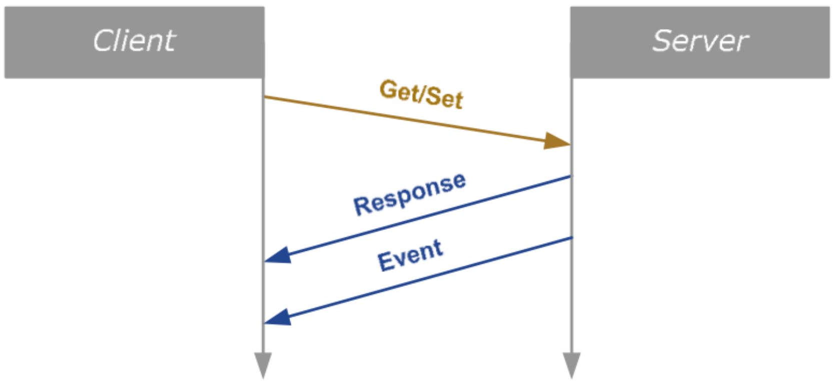


Figure 4.8: Field

This structure supports both **synchronous** (critical for control) and **asynchronous** (typical for infotainment or ADAS) communications, providing maximum flexibility.

Compatibility, transport, and segmentation SOME/IP uses TCP or UDP as transport protocol over Ethernet and supports both IPv4 and IPv6. This ensures compatibility with legacy and modern networks, promoting interoperability and integration in distributed architectures.

Furthermore, in the case of very large messages, the protocol implements UDP packet segmentation logic, avoiding fragmentation at the IP level.

Service discovery and Pub/Sub The **SOME/IP-SD** (Service Discovery) protocol is responsible for dynamic service discovery and is the main part of SOME/IP.

The **server** can:

- send `OfferService` via multicast to announce services;
- respond to `FindService` requests from clients, in case the client has not received the OfferService or explicitly requests a service;
- announce the deletion of services via `StopOffer`.

The **Subscribe** mechanism allows **clients** to receive only the events or groups of events they are interested in, via messages:

- `Subscribe`;
- `SubscribeAck`, i.e., confirmation by the server that the subscription has been successful;
- `SubscribeNack`, i.e., rejection by the server due to service unavailability;
- `StopSubscribe`, i.e., when the client cancels the subscription to an event.

Packet structure of SOME/IP SOME/IP messages use a structured format designed for interoperability between heterogeneous systems. A typical SOME/IP message, illustrated in Figure 4.9, contains an header that includes [23]:

- **Service ID** – uniquely identifies the service;
- **Method/Event ID** – identifies the specific method or event;
- **Client ID** - unique identifier for the calling client inside the ECU;
- **Session ID** – identifier for session handling;
- **Length** – contain the length in byte starting from Request ID/Client ID until the end of the SOME/IP message;
- **Protocol version** and **Interface version**;
- **Message Type** – specifies the type (request, response, notification, etc.);

- **Return Code** - specifies the result of the request or indicates the type of error that occurred during its processing.

The payload, serialized according to middleware rules, follows the header.

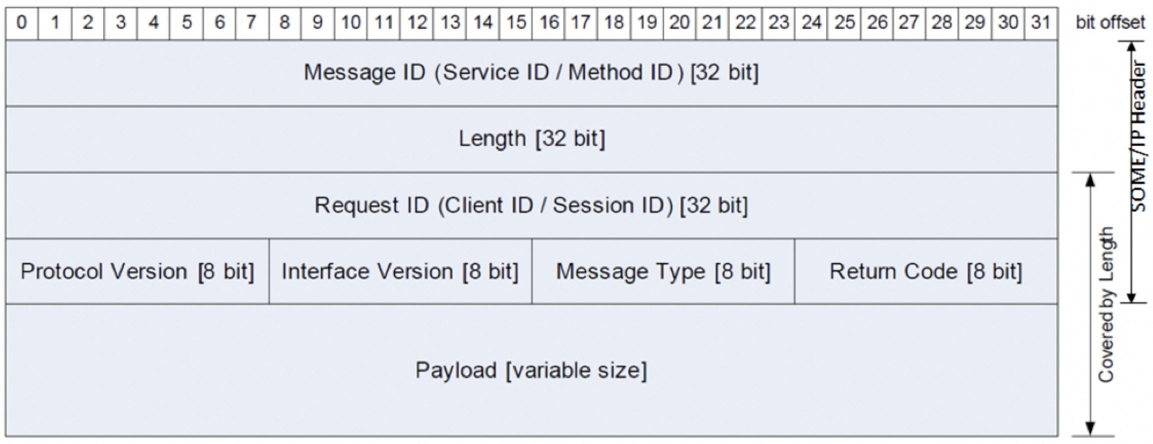


Figure 4.9: SOME/IP header format [23]

Service Discovery (SD) messages, illustrated in Figure 4.10, are used to dynamically advertise and locate services. These shall be sent over UDP and composed of [24]:

- **Service ID**, always set to 0xFFFF, and **Method ID** to 0x8100, identifying the message as a Service Discovery packet;

- **Protocol Version** and **Interface Version** are both set to 0x01, indicating the standard version of the SD protocol;

- **Message Type** is set to 0x02, which stands for a *Notification*;

- **Return Code** is set to 0x00, meaning that no error occurred;

- **Client ID** is always 0x0000 because only one SD instance exists;

- **Session ID** is an identifier for session handling. It must never be zero;

- **Entries** (such as `OfferService`, `FindService`, `Subscribe`), with the corresponding length;

- **Options**, with the corresponding length, are used to transport additional information to the entries. This includes for instance the information how a service instance is reachable (IP address, Transport Protocol, Port Number).

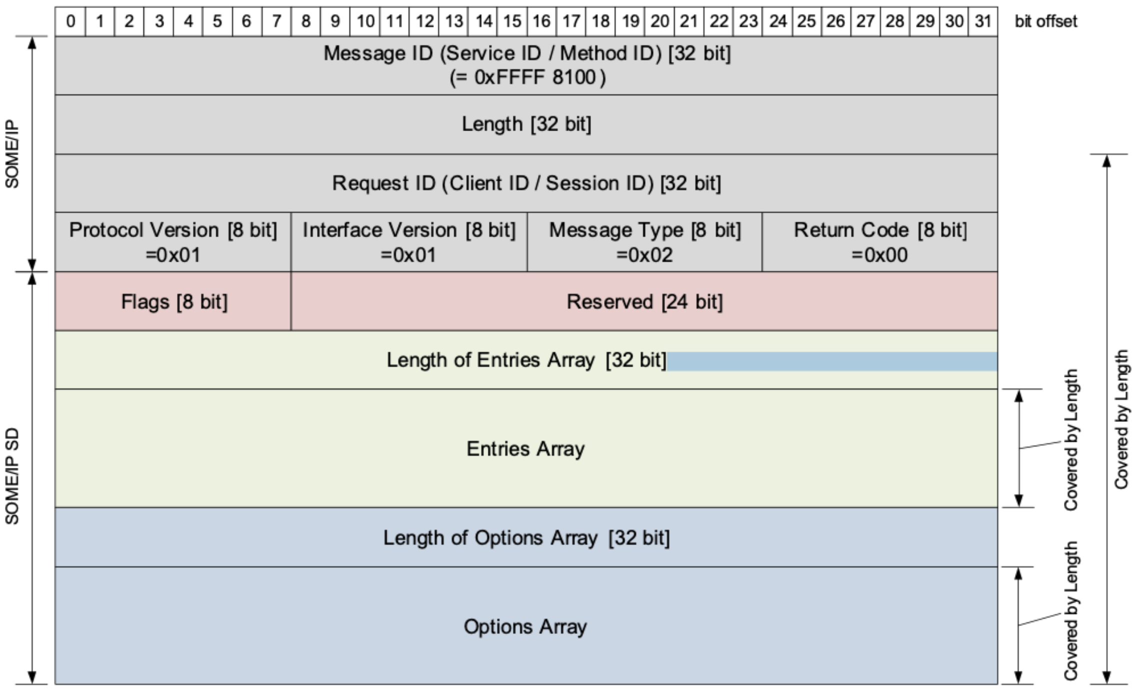

bit offset
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Message ID (Service ID / Method ID) [32 bit] (= 0xFFFF 8100)
Length [32 bit]
Request ID (Client ID / Session ID) [32 bit]
Protocol Version [8 bit] =0x01
Interface Version [8 bit] =0x01
Message Type [8 bit] =0x02
Return Code [8 bit] =0x00
Flags [8 bit]
Reserved [24 bit]
Length of Entries Array [32 bit]
Entries Array
Length of Options Array [32 bit]
Options Array
SOME/IP
SOME/IP SD
Covered by Length
Covered by Length
Covered by Length

Figure 4.10: SOME/IP-SD header format [24]

This format allows efficient real-time negotiation and setup of service-based communication.

Advantages in the automotive context The advantages of the SOME/IP approach are:

- **High performance**: ability to handle large volumes of data with low latency.
- **Reliability**: error handling and message receipt confirmation.
- **Scalability**: suitable for networks with few or many ECUs, making it the ideal protocol for use in complex automotive systems.
- **Efficiency**: multicast support to reduce traffic.
- **Standardization**: compliant with AUTOSAR Adaptive, ensuring interoperability between different automotive manufacturers and suppliers.
- **Flexibility**: independent of data format.
- **Security**: support for message encryption and authentication.

Practical example of use In a modern vehicle, when the driver enters the destination, the navigation system's ECU can send turn-by-turn instructions to the digital cluster via SOME/IP messages. This ensures that the driver sees the directions on the dashboard. The same data can also be used by the infotainment system to provide additional information to the user, such as points of interest or real-time traffic. This type of integrated and asynchronous communication demonstrates the protocol's ability to provide advanced user experiences and system modularity [17].

Implementation with vSomeIP In this project, **vSomeIP**, an open source implementation in C++ based on SOME/IP protocol, used in AUTOSAR Adaptive environment and developed by COVESA (formerly GENIVI) [2], was used.

It provides:

- API for defining and managing servers and clients;
- serialization and publication of events.
- discovery and subscription management;

Thanks to its compatibility with AUTOSAR and the modularity of its approach, vSomeIP made possible, in this project, to move from ARXML models exported from PREEvision to a concrete implementation in testable C++ code.

SOME/IP is now a key element in the transition to SDVs and zonal E/E architectures. Supporting **dynamic communication**, **interoperability** and **standardization**, it represents a robust and scalable **middleware** for the integration of distributed services in the intelligent vehicles of the future.

The theoretical concepts, technologies, and tools described in this chapter form the foundation on which the developed project was built. The next chapter will present the practical implementation of a SOA architecture based on PREEvision and SOME/IP, highlighting the design choices, challenges encountered, and results obtained.

[2]The Connected Vehicle Systems Alliance, previously known as the GENIVI Alliance, is a non-profit automotive industry alliance that develops reference approaches for the integration of operating systems and middleware in connected vehicles and related cloud services.

Chapter 5

Modeling a Service-Oriented Architecture with PREEvision

This chapter describes in detail the architectural modeling process carried out using PREEvision. The model created represents the entire software and hardware architecture of a distributed system based on the Service-Oriented architecture paradigm, in accordance with AUTOSAR standards. The modeling flow, which is structured and hierarchical, covers all the fundamental phases of development: it starts with the definition of functional requirements and then describes the software and service architecture. Next, the hardware architecture is illustrated, with particular attention to the mapping between software and hardware levels, and the analysis of the communication level concludes the process.

5.1 Adopted development approach

In the project described in this thesis, a model-based engineering approach (Model-Based Systems Engineering) was adopted, as described in Section 3.4, with the aim of designing a **Service-Oriented architecture** conforming to the **AUTOSAR Adaptive** standard and implementing it in a simplified **zonal network topology**.

This approach made it possible to address the growing complexity of modern vehicle systems by following a structured, traceable, and automatable flow, in line with the needs of the contemporary automotive industry.

The complete workflow followed for the design, modeling, and implementation of the architecture is shown in Figure 5.1.

The process is divided into two main phases: a first phase dedicated to architectural modeling using the **PREEvision** tool, and a second phase of implementation in **C++** language, using the `vSomeIP` middleware, and run-time verification on embedded boards. In particular, in the initial phase, both a software-oriented system, based on traditional functional control units, and a service-oriented system, in which the functionalities are implemented as interoperable services on an Ethernet network, were modeled to see the significant differences.

The board used for practical validation is the **miriac® SBC-S32G274A** from *MicroSys*, an embedded board based on the *NXP S32G274A* automotive processor, with TSN Ethernet support, Cortex-M7 real-time core, and CAN, LIN, and FlexRay interfaces. It is ideal for testing SOA architectures on Ethernet networks thanks to native support for SOME/IP and compatibility with safety-critical environments [19].

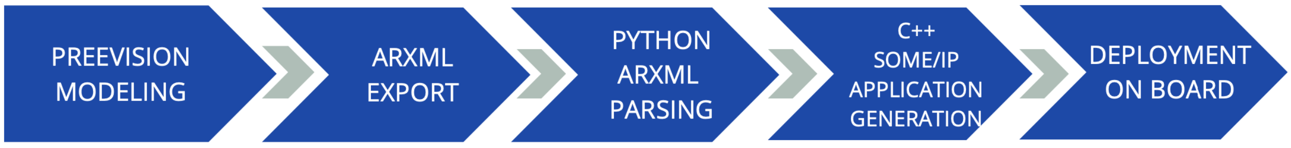


Figure 5.1: Project workflow

5.2 Modeling in PREEvision

For the modeling phase, PREEvision was used, one of the most comprehensive MBSE tools for E/E engineering in the automotive sector, already discussed in Section 4.1. The tool made it possible to create and compare two different architectural configurations:

- a first **software-oriented** architecture, in which the functionalities are implemented through software components directly linked to specific ECUs, typical of traditional architectures;
- a second **service-oriented** architecture, in which the functionalities are decoupled from the hardware and implemented as services according to the AUTOSAR Adaptive standard, with communication based on SOME/IP.

Both configurations were modeled and mapped onto a simplified Ethernet network reflecting a zonal logic.

The modeling performed in PREEvision followed a well-defined flow, divided into the following four macro-phases:

1. **Requirements definition**: the functional requirements of the system were described using the *Requirements Management* module.

2. **Software design and service definition**: the software components were modeled with the various inputs and outputs according to the defined requirements. Subsequently, following the Service-Oriented approach, the services were defined with their interfaces. In particular, the client and server entities, methods, events, and fields to be communicated via the SOME/IP protocol were specified.

3. **Mapping to ECUs and network topology**: each software component or service provider/consumer was assigned to a specific ECU within a logical Ethernet topology.

4. **Export of ARXML files**: finally, the complete configurations were exported in ARXML format for use in the implementation phase with `vsomeip`.

This pipeline minimized the gap between design and implementation, improving traceability, consistency between layers, and the quality of the resulting code.

The following paragraphs describe this phases and analyze in detail the two architectural models developed (Software-Oriented and Service-Oriented), highlighting the main differences, design choices, and benefits obtained from the transition to the SOA paradigm.

5.3 Requirements definition

As a starting point for designing the Service-Oriented architecture, the functional requirements for four main services were defined and modeled. Although simpler than in a real system, these services were chosen as representative examples of functionality that could actually be present in a modern vehicle E/E architecture.

The goal was not to replicate the full complexity of a real system, but to design a modular and coherent structure that could serve as a foundation for demonstrating the MBSE flow, ensuring multi-level consistency within the model, and enabling seamless integration with the AUTOSAR Adaptive standard — with the potential to scale to larger architectures in future developments.

The selected services are:

- **Alternative Route Service** – Calculates an alternative route for the driver based on specific metrics.

- **Driving Monitoring Service** – Monitors driver behavior and sends alerts if the attention level is low.

- **Driving Hours Monitoring Service** – Monitors and calculates the driver's driving hours and is linked to the Driving Monitoring Service.

- **Traffic Condition Service** – Analyzes traffic conditions and categorizes them as 0 (low), 1 (medium), or 2 (high).

All functional requirements related to the modeled services have been formally defined and reported in PREEvision using the *Requirements Management* module. This tool allows each requirement to be associated with **key attributes** such as hierarchical level, technical description, requirement type, and direct links to other architectural elements (functions, software components, interfaces, etc.), ensuring complete traceability throughout the development cycle.

In the context of the project, requirements were classified into three main categories:

- **Requirement (shall)**: technical requirements that the system *should* meet to ensure proper functioning. For example: "The position of the vehicle should be updated at least every 5 seconds". These requirements are verifiable, but in some cases may allow for tolerances.
- **Definition (must)**: binding requirements that the system *must* meet without exception, as they are related to physical, normative, or safety constraints. For example: "The vehicle must be equipped with front airbags for the driver and passenger". Failure to comply with a "must" requirement would compromise the validity of the system.
- **Information (can)**: requirements of a descriptive or informative nature, that the system *can* fulfill to enhance quality, usability, or user experience. For example: "The interior lights of the vehicle may turn on automatically when the doors are opened".

This categorization, adopted directly within PREEvision, has made it possible to clearly distinguish between mandatory, desirable, and accessory requirements. In addition, each requirement has been linked to its respective functions or components, establishing a direct link between the initial requests and the actual implementations, according to the top-down traceability principles of the Model-Based approach.

Below is an example of requirements gathering for the **Traffic Condition Service** [Table 5.1].

Table 5.1: Example of Requirements for Alternative Route Service

Level	**Text**	**Type**
1-1	**1 AlternativeRoute**	Req Package
2-2	**1.1 DataCollectionRealTime**	Req Package
3-3	1.1.1 Radar	Is Heading
4-4	It should measure the distance and speed of other vehicles around, to detect slow traffic situations to suggest alternative route	Req (shall)
5-4	The radar must have a detection range of at least 100 meters and an update rate of at least 10 Hz	Definition (must)

Level	Text	Type
6-3	1.1.2 Camera	Is Heading
7-4	It should detect the presence of traffic, obstacles or road works in real time to suggest a change of route	Req (shall)
8-3	1.1.3 GPS	Is Heading
9-4	It should detect the current position of the vehicle and update it continuously (every second) to ensure accurate alignment of the suggested route with the actual position	Req (shall)
10-4	The GPS module must guarantee an accuracy of at least ± 3 meters and support real-time positioning updates at 1 Hz or higher	Definition (must)
11-3	1.1.4 InfoV2V	Is Heading
12-4	The system need to integrate data from nearby vehicles, such as speed and location, to get an up-to-date view of the traffic	Req (shall)
13-4	Vehicles must be equipped with V2V-compatible communication devices to exchange data on speed, position and surrounding situation	Definition (must)
14-3	1.1.5 Map Integration	Is Heading
15-4	It should align updated data from online maps to check for road congestion, roadworks and accidents. It should calculate all possible route options from the current location to the destination, highlighting alternate routes in case of problem along the main route	Req (shall)
16-4	The system must use a map provider with real-time traffic updates and route recalculation APIs accessible via HTTPS	Definition (must)
17-3	1.1.6 BackendInfo	Is Heading
18-4	It should access stored traffic history to identify usual congestion areas and suggest routes that typically remain less affected	Req (shall)
19-4	The backend must store at least 30 days of historical traffic data, indexed by time, location and road segments	Definition (must)
20-2	**1.2 DataAnalysis**	Req Package
21-3	1.2.1 RealTimeUpdated	Is Heading
22-4	The system should combine data from GPS, cameras, radar and V2V to continuosly update traffic status, automatically recalculating the alternative route if traffic conditions change	Req (shall)
23-3	1.2.2 SpeedAnalysis	Is Heading

Level	**Text**	**Type**
24-4	It should monitor the average speed of both the vehicle and surrounding vehicles to identify slow-down or congestion situations and update the suggested route based on this information	Req (shall)
25-3	1.2.3 BackendIntegration	Is Heading
26-4	The system should be able to access alternative routes already calculated by the backend and should receive real-time updates on traffic conditions from the cloud, which may affect the suggested routes	Req (shall)
27-3	The system can take into account seasonal traffic variations (e.g., holidays, school closures, typical summer congestion) to refine route suggestions	Info (can)
28-2	**1.3 Output**	Req Package
29-3	The output is the definition of an alternative route possibly with an audible signal on the infotainment system	Req (shall)

5.4 Software architecture

After defining the requirements, the software architecture was modeled following a functional scheme inspired by the **AUTOSAR Classic** paradigm.

The software architecture was modeled entirely within the graphical environment provided by PREEvision. In this phase, the application software components and their logical connections were defined, independently of their physical implementation and assignment to the ECUs.

As shown in Figure 5.2, the design flow adopted in PREEvision for exporting the *ARXML* file follows a well-defined sequence of steps, starting with the modeling of the **software and hardware architecture**. Once these two levels have been defined, the **software components are mapped to the physical ECUs** explicitly establishing on which control unit each component will be executed.

Once the mapping is complete, PREEvision allows automatic **configuration of communication** between components via *data elements*, which describe the information exchanged by the logic blocks. The *signal router*, a function integrated into the tool, automatically generates the signals and their transmissions based on the mappings made and configures the routing for the network. This makes it possible to design communication on CAN, LIN, FlexRay, or Ethernet buses, ensuring consistency between the software and physical levels.

All these steps are preparatory to the export of ARXML files, which represent the standard AUTOSAR artifacts that can be used in subsequent low-level software development phases.

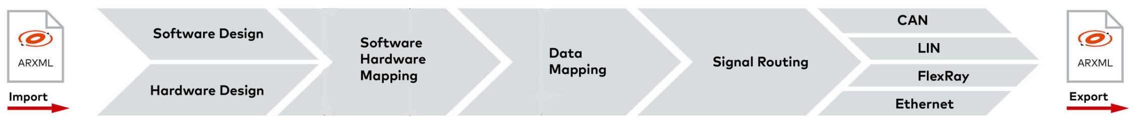


Figure 5.2: System and software design process in PREEvision [15]

The model was not imported from *ARXML* files, but the modeling process was carried out entirely from scratch. Furthemore, in this phase, the generation of physical signals, such as those on CAN buses, was not included, as this had already been addressed in previous theses by Luca Valentini and Giovanni Musto and was not central to the objectives of this work. The focus was on the evolution towards a service-oriented architecture: as will be seen in the following paragraphs, only in the Service-Oriented context have communication signals been effectively modeled, in particular using the SOME/IP protocol.

Within this workflow, one of the central activities is the **software design** phase, which is described exclusively in this section, while hardware design and mapping to physical nodes will be covered in subsequent paragraphs.

Each **Application Software Component** has been defined as a functional block equipped with communication ports (*Sender Port* and *Receiver Port*), through which it exchanges data with other components or with virtual peripherals. The ports are in turn associated with interfaces defined by **Sender/Receiver Interface (SRI)**, which describe how the components interact. Each interface is characterized by one or more **Data Elements (DE)**, which represent the information entities exchanged within the system and are associated with a data type (*Application Data Type*).

Sensor and Actuator SW Components were used to represent the input and output points of the system, simulating the acquisition of signals by sensors or the activation of actuators, respectively. In this context, a sensor typically has only sender ports, while an actuator has only receiver ports.

Figure 5.3 shows a concrete example of the **Traffic Condition Service** modeled in PREEvision, in which are visible:

- the Application Software Component;
- the connections via SRI interfaces;
- the associated Data Elements;
- the mapping with the previously defined functional requirements.

In this example, the main application component is the **Alternative Route Controller**, which is responsible for the route recalculation logic.

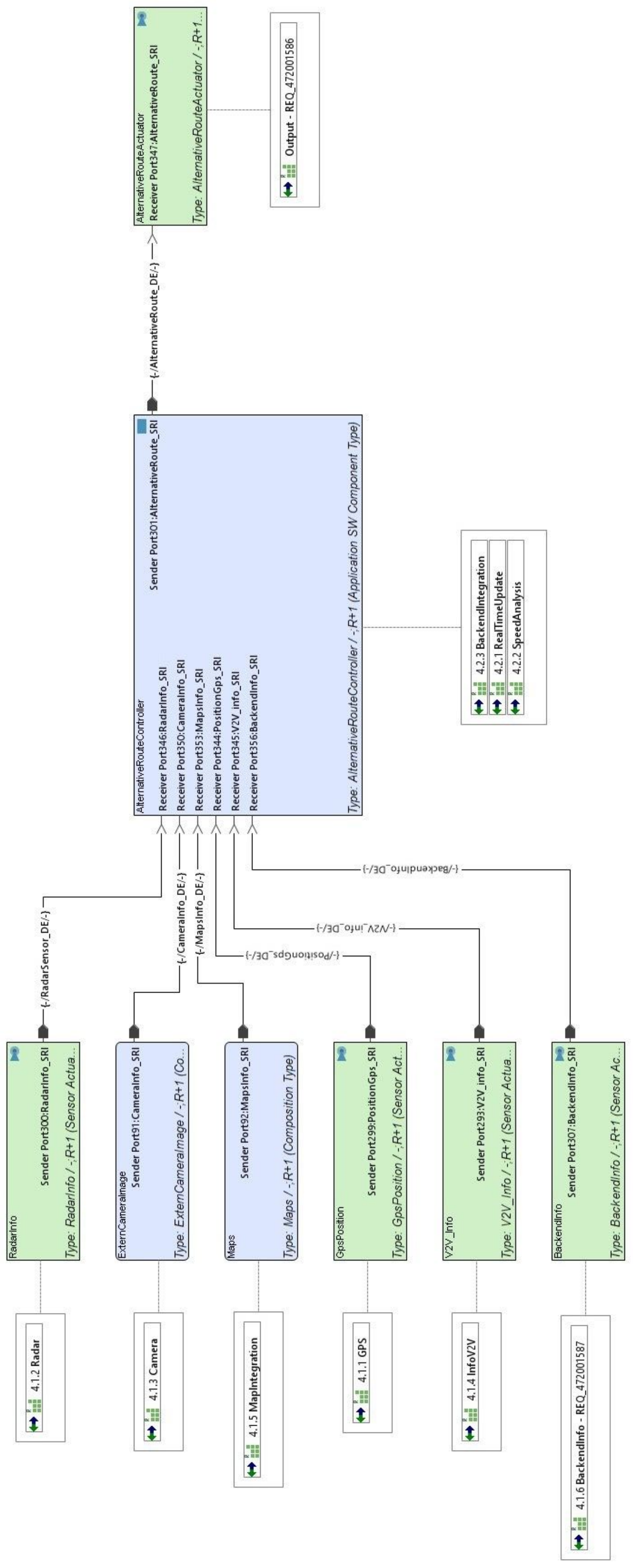
Among the inputs, there is, for example, a **Sensor Software Component** called *Radar Info*, modeled to simulate the collection of environmental information. This component sends data via a **Sender Receiver Interface** `RadarInfo_SRI`, which is associated with the **Data Element** `RadarSensor_DE`. The latter represents the information value transmitted to the controller, which uses it as logical input.

At the output, the **Alternative Route Controller** produces a result through the `AlternativeRoute_SRI` interface, connected to the **Data Element** `Alternative Route_DE`, which transmits the information to an **Actuator Software Component** called *Alternative Route Actuator*. The latter represents the block responsible for managing the command to the physical system, thus completing the functional chain from the sensor to the actuator.

Each component has been graphically linked to the others in a coherent data flow, and the overall structure has been kept aligned with the requirements defined above (Section 5.3). Furthermore, direct mapping between functional blocks and requirements is not mandatory but is extremely useful for ensuring design consistency and end-to-end traceability.

Finally, it should be noted that the focus of PREEvision in this phase is on **architectural design** — and not code implementation — according to the AUTOSAR methodology. The generated artifacts, such as ARXML files, can then be used in external environments for actual software development.

All design choices made in the context of this modeling have been deliberately oriented towards **simplicity**, with the aim of focusing on the **methodological process** of building the architecture, rather than on the exact reproduction of the behavior of a specific vehicle service.



AlternativeRouteActuator
Receiver Port347:AlternativeRoute_SRI
Type: AlternativeRouteActuator / -;R+1...
Output - REQ_472001586
{-/AlternativeRoute_DE/-}
AlternativeRouteController
Sender Port301:AlternativeRoute_SRI
Receiver Port346:RadarInfo_SRI
Receiver Port350:CameraInfo_SRI
Receiver Port353:MapsInfo_SRI
Receiver Port344:PositionGps_SRI
Receiver Port345:V2V_info_SRI
Receiver Port356:BackendInfo_SRI
Type: AlternativeRouteController / -;R+1 (Application SW Component Type)
4.2.3 BackendIntegration
4.2.1 RealTimeUpdate
4.2.2 SpeedAnalysis
{-/RadarSensor_DE/-}
{-/CameraInfo_DE/-}
{-/MapsInfo_DE/-}
{-/PositionGps_DE/-}
{-/V2V_info_DE/-}
{-/BackendInfo_DE/-}
RadarInfo
Sender Port300:RadarInfo_SRI
Type: RadarInfo / -;R+1 (Sensor Actua...
ExternCameraImage
Sender Port91:CameraInfo_SRI
Type: ExternCameraImage / -;R+1 (Co...
Maps
Sender Port92:MapsInfo_SRI
Type: Maps / -;R+1 (Composition Type)
GpsPosition
Sender Port299:PositionGps_SRI
Type: GpsPosition / -;R+1 (Sensor Act...
V2V_Info
Sender Port293:V2V_info_SRI
Type: V2V_Info / -;R+1 (Sensor Actua...
BackendInfo
Sender Port307:BackendInfo_SRI
Type: BackendInfo / -;R+1 (Sensor Ac...
4.1.2 Radar
4.1.3 Camera
4.1.5 MapIntegration
4.1.1 GPS
4.1.4 InfoV2V
4.1.6 BackendInfo - REQ_472001587

Figure 5.3: Alternative route service software diagram

5.5 Service architecture

After creating a first software model explained in the previous section, modeling continued following the Service-Oriented paradigm.

This second phase was also developed in PREEvision, leveraging the features of the AUTOSAR Adaptive standard, designed to support SDVs.

The main difference between the two models lies in the level of abstraction and modularity. In the first case, the software blocks (sensor, application, and actuator components) are connected via *sender* and *receiver* ports, defined by **S/R interfaces** and **Data Element**. Each connection represents a direct flow of data that is exchanged via punctual and predefined signals, typical of static (signal-based) communication.

In the second case, however, the system is modeled as a set of **services**, each of which is offered by a *Service Provider* and can be requested by one or more *Service Consumers*. Services (such as GPS, Radar, Maps, V2V, Historical Data) are identified by specific interfaces and instances, which allow them to be dynamically identified via **Service Discovery**, and communication is managed in a dynamic client-server model managed via middleware such as **SOME/IP**.

The service-oriented architecture ensures a high degree of decoupling between components, promotes the reusability and updatability of services, and enables flexible and scalable interaction between ECUs as discussed in previous chapters.

This architectural evolution reflects the shift from rigid embedded systems to more dynamic and reconfigurable vehicle systems.

Figures 5.3 and 5.4 show the different architectural modeling of the two cases.

In the first model (Figure 5.3), as explained in the previous section, each block has a specific role (e.g., sensor, actuator, controller) and is subsequently mapped to ECUs or physical peripherals, thus defining a direct coupling between software logic and hardware implementation.

In the second model (Figure 5.4), on the other hand, components act as providers or consumers, and interactions happen through *Service Interfaces*, within which RPC methods, asynchronous events, or accessible fields are defined. People no longer think in terms of signals or direct connections, but in terms of services offered and requested. Coupling no longer occurs between software and ECUs, but between providers and consumers, which increases the **modularity**, **updatability**, and **reusability** of the software.

The figure illustrates, as in the previous case, the **Alternative Route** function. However, while in the first scenario this functionality was handled by a centralized controller component (*Alternative Route Controller*) implementing the logic through clearly defined input/output flows, in this case it is modeled as a distributed service (*Alternative Route Service*).

In the traditional approach, inputs and outputs follow a predetermined and

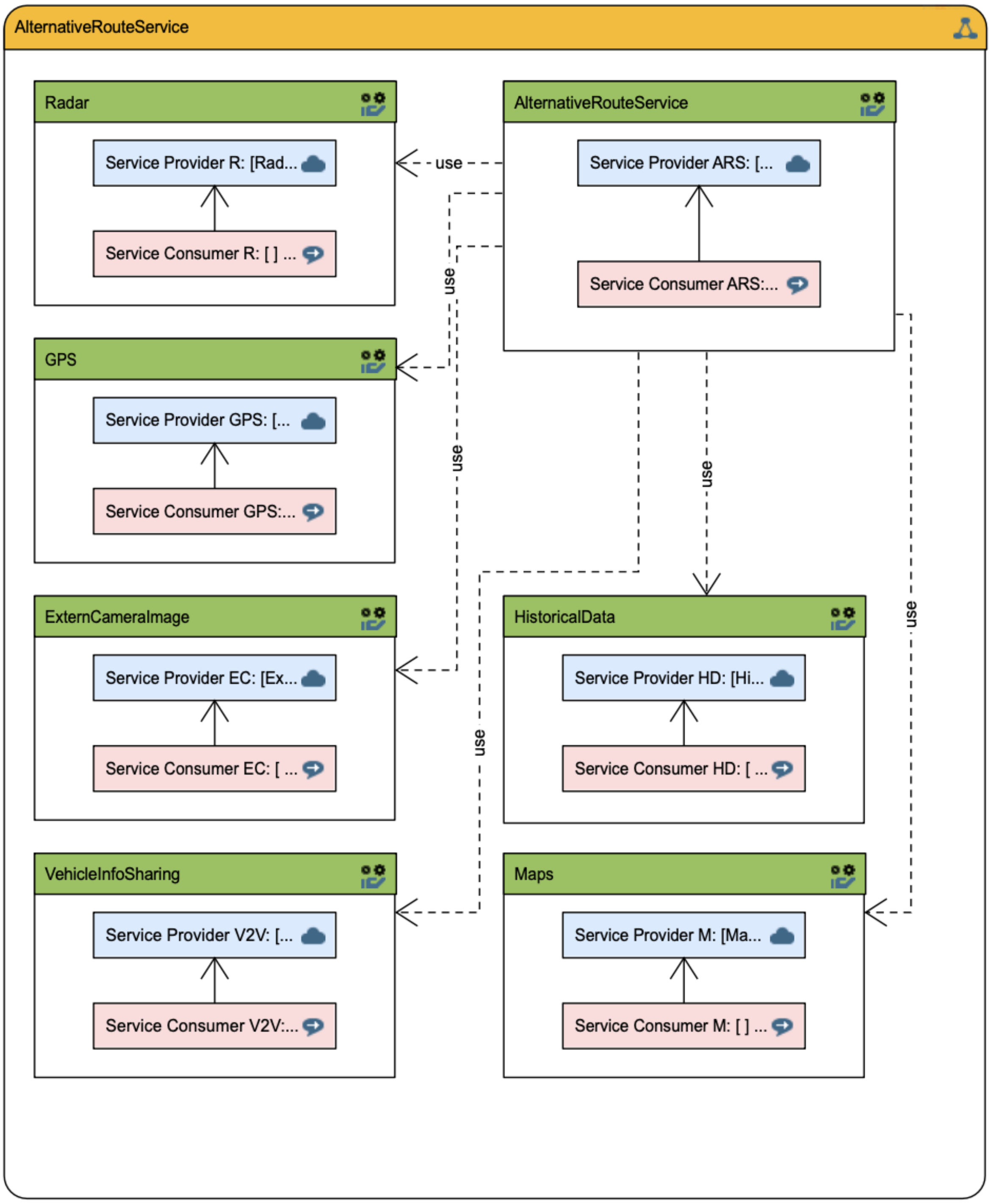

AlternativeRouteService
Radar
Service Provider R: [Rad...
Service Consumer R: [ ] ...
AlternativeRouteService
Service Provider ARS: [...
Service Consumer ARS:...
use
GPS
Service Provider GPS: [...
Service Consumer GPS:...
ExternCameraImage
Service Provider EC: [Ex...
Service Consumer EC: [...
HistoricalData
Service Provider HD: [Hi...
Service Consumer HD: [...
VehicleInfoSharing
Service Provider V2V: [...
Service Consumer V2V:...
Maps
Service Provider M: [Ma...
Service Consumer M: [ ] ...

Figure 5.4: Alternative route service service diagram

centralized flow. In contrast, the service-oriented representation does not rely on a single I/O flow but is composed of multiple interacting services, each defined by a provider and consumer communicating with each other.

In particular, it is possible to see how, for example, the *Radar* function in the first case is represented by a sensor block equipped with a sender port connected directly to a receiving application component. Communication takes place via a *Data Element* and an *S/R interface* and represents a static point-to-point connection defined during the design phase.

In the service-oriented model, the same functionality is implemented as a service, accessible through a *Service Interface*. This service is offered by a provider (*Service Provider R*) and can be used by one or more consumers. The use of the service is represented graphically by the "use" arrow in PREEvision, which clearly shows how the service can be shared and reused by different components.

As in traditional software modeling, it was also possible to map the functional requirements. However, in this model, the link was established at the service level, highlighting the link between each planned functionality and the service that implements it. End-to-end traceability and consistency in the design process is therefore also guaranteed in this case, as shown in Figure 5.5. In particular, the figure shows the mapping of the *GPS service*, but this has been done with all the other services present.

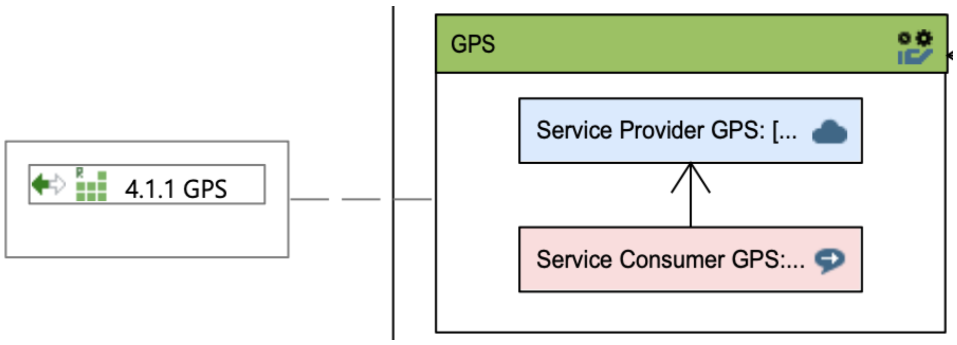


Figure 5.5: Requirement GPS mapping

The workflow followed to model the entire service-oriented architecture is illustrated in Figure 5.6. It starts with the abstract definition of the services and ends with the configuration of Ethernet communication, thanks to PREEvision's *SOA and Ethernet Explorer*.

In this case too, we did not start by importing the ARXML, but modeled the architecture from scratch.

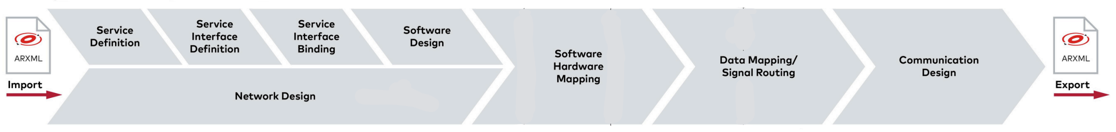

ARXML
Import
Service Definition
Service Interface Definition
Service Interface Binding
Software Design
Network Design
Software Hardware Mapping
Data Mapping/ Signal Routing
Communication Design
ARXML
Export

Figure 5.6: Service-Oriented modeling workflow in PREEvision [15]

The nine main phases that make up the process are detailed below.

1. **Service Definition**: In this initial phase, the services offered by the system are defined. Each service represents a logical functionality that can be provided by a provider and used by one or more consumers. For example, the services already described in Section 5.3 have been modeled in the project:

   - *Traffic Condition Service*
   - *Alternative Route Service*
   - *Driving Hours Monitoring Service*
   - *Driving Behavior Monitoring Service*

   These main services, which represent comprehensive application features aimed at vehicle behavior, have been supplemented by numerous auxiliary services, such as Radar, GPS, External Camera, etc.

   These services provide data or specific information that is consumed by the main services, according to a modular and reusable logic. For example, both the *Traffic Condition Service* and the *Alternative Route Service* use the *Radar Service* as a source of environmental data. In Figure 5.4 it is possible to see this interaction between the *Alternative Route Service* and the *Radar Service.*

2. **Service Interface Definition**: Each service is associated with an interface, i.e., an API between the provider and the consumer that describes how the consumer can use the service. Interfaces include:

   - **Methods**: i.e., functions that are executed by providers upon request from one or more consumers.

- **Events**: i.e., an update to the data. The provider decides when to send these updates to one or more consumers who have subscribed to receive such notifications.
- **Properties**: i.e., fields exposed by providers to one or more consumers through methods such as `get()` and `set()`, if provided. Consumers can also optionally receive notifications if the value of a field changes.

An example of **Service Interface** modeled in this project is represented by the `Alternative Route Service_I` and `Driver Behavior Monitoring Service_I` interfaces, shown in Figure 5.7, and respectively associated with the *Alternative Route Service* and *Driver Behavior Monitoring Service*.

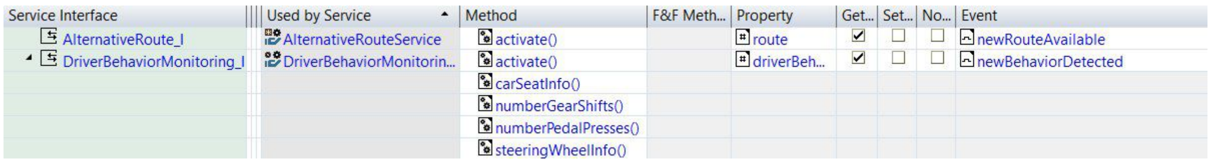
For the *Alternative Route Service*, the interface provides:

- a `activate()` method for activating the service;
- a *property* named `route`, accessible via the `get()` method;
- an asynchronous event `new Route Available` to notify the availability of a new route.

The `Driver Behavior Monitoring Service_I` interface, on the other hand, is more complex and includes:

- the `activate()` method to start monitoring;
- `carSeatInfo()`, which should return information from a sensor located on the seat, if present;
- `numberGearShifts()`, which should detect how many times the driver has changed gear, useful for evaluating their level of activity;
- `numberPedalPressed()`, which should monitor the frequency of pedal pressure;
- `steeringWheelInfo()`, which should provide data derived from a sensor on the steering wheel, if present;
- a *property* called `driverBehavior`, accessible via `get()` method;
- a `new Behavior Detected` event, emitted whenever a new behavior pattern is detected.

All interfaces defined in this project share a common method called `activate()`. This method should allow the associated service to be dynamically activated or deactivated, promoting system modularity.



Service Interface	Used by Service	Method	F&F Meth...	Property	Get...	Set...	No...	Event
AlternativeRoute_I	AlternativeRouteService	activate()		route	✓	☐	☐	newRouteAvailable
DriverBehaviorMonitoring_I	DriverBehaviorMonitorin...	activate()		driverBeh...	✓	☐	☐	newBehaviorDetected
		carSeatInfo()						
		numberGearShifts()						
		numberPedalPresses()						
		steeringWheelInfo()						

Figure 5.7: Service interfaces

3. **Service Interface Binding**: In this phase, the transport layer of the middleware is configured. In order for the middleware to be able to recognize method calls and correctly transmit event information, each **Service Interface** and its elements must be uniquely identified by an ID so that messages can be correctly routed on the bus.

   In the context of AUTOSAR Adaptive, the middleware currently supported is exclusively **SOME/IP**.

   In this phase, it is also possible to create **Event Groups**. These are used to reduce communication overhead for subscribing to an event. A client subscribes to a group of events, i.e., a logical grouping of events offered together. Event Groups also require unique IDs.

4. **Software Design**: In AUTOSAR Adaptive, the software architecture is described by **Adaptive Application SW Components**. The Adaptive Application SW Components implement the Service Interfaces. How to group Service Interfaces in software components depends on how we model the services. A server and a client software component are necessary for each Service Interface.

   In this scenario, **Client/Server** ports replace traditional S/Rs, as we can see in Figure 5.8. A client component can access the methods offered by a server component through a **Service Interface**. In this context, a *client port* makes requests to the service, while a *server port* provides the implementation of the service itself. The interfaces associated with these ports no longer contain simple *Data Elements* as in the Classic approach, but are structured according to the service paradigm, including elements such as methods, events, and fields, in line with the AUTOSAR Adaptive architecture.

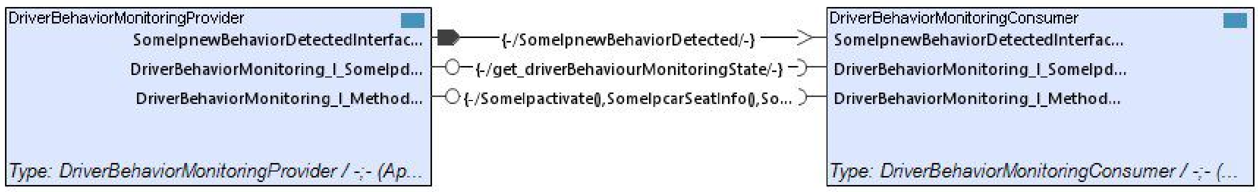

DriverBehaviorMonitoringProvider
SomeIpnewBehaviorDetectedInterfac...
DriverBehaviorMonitoring_I_SomeIpd...
DriverBehaviorMonitoring_I_Method...
Type: DriverBehaviorMonitoringProvider / -;- (Ap...
{-/SomeIpnewBehaviorDetected/-}
{-/get_driverBehaviourMonitoringState/-}
{-/SomeIpactivate(),SomeIpcarSeatInfo(),So...
DriverBehaviorMonitoringConsumer
SomeIpnewBehaviorDetectedInterfac...
DriverBehaviorMonitoring_I_SomeIpd...
DriverBehaviorMonitoring_I_Method...
Type: DriverBehaviorMonitoringConsumer / -;- (...

Figure 5.8: Software design SOA

Figure 5.8 shows the software components that implement the **provider** and **consumer** of the *Driver Behavior Monitoring Service*. How it can be seen, communication between the two is modeled using two distinct types of ports.

The **methods** and **properties** of the Service Interface, such as `activate()`, `getDriverBehavior()` or `carSeatInfo()`, are managed through **client/server**

ports. These ports model synchronous communication, in which the client component invokes a method or reads/writes a property offered by the server, waiting for a response.

In addition to these, there are also **sender/receiver ports**, which are associated with **event** management. These ports allow asynchronous transmission of notifications, such as the `new Behavior Detected` event, from the service provider to the service consumer. This is because an event can be notified to multiple recipients simultaneously, without waiting for a response.

5. **Network Design**: this phase involves the definition of the network infrastructure between ECUs and will be analyzed in detail in the next section.

6. **Software-Hardware Mapping**: Each software component (provider or consumer) is assigned to an ECU unit on the network. In the Section 5.6.1 there are further details.

7. **Data Mapping and Signal Routing**: With the modeling of services and software interfaces on one side, and the definition of the hardware network topology along with the mapping of software components to ECUs on the other, the next step enabled by PREEvision is the configuration of **signal routing**. This phase allows for the automatic generation of signals to be transmitted over communication buses and the mapping of *data elements/operations* to the corresponding signals, which is discussed in detail in Section 5.7.

   This enables the design of communication over various bus systems, including CAN, LIN, FlexRay, and Ethernet. In this project, the focus is specifically on Ethernet communication using the SOME/IP protocol.

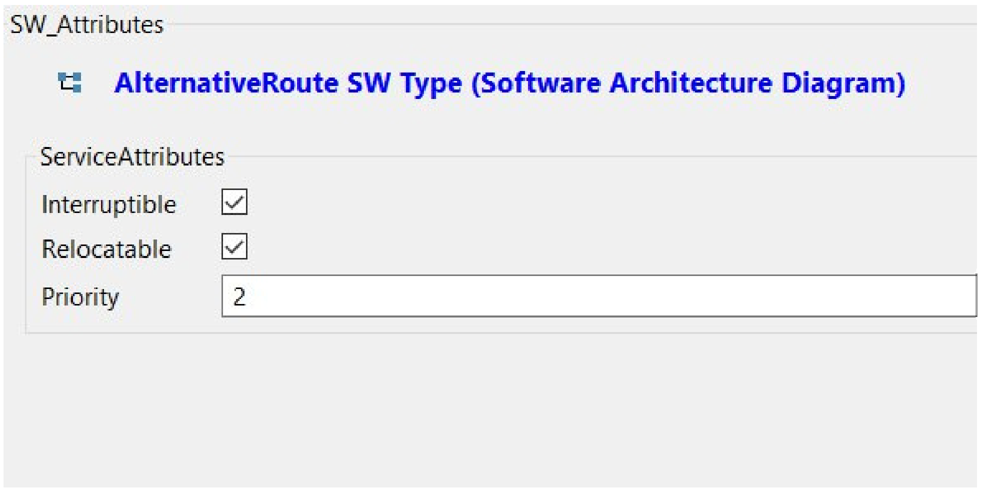
8. **Communication Design**: The Ethernet network is modeled with IP configuration, multicast/unicast, TCP and UDP ports. The socket addresses are defined for each provider/consumer communication. See section 5.7 for more details.

9. **ARXML Export**: Finally, PREEvision allows the entire modeled architecture to be exported in *ARXML* format, including descriptions of services, interfaces, software components, and network configurations. These files can then be used during implementation, as was done in this project, which we will see later in Chapter 6.

This service-oriented modeling has enabled a modular, reusable, and scalable representation of the vehicle architecture with a **high level of abstraction and consistency**. Furthermore, the transition from S/R to C/S ports represents one of the fundamental changes between AUTOSAR Classic and Adaptive. Ports are no longer simple “signal conduits,” but define functional roles within a **distributed** service-oriented architecture.

5.5.1 Custom attributes definition and dynamic service shifting

One of the innovative aspects of this work lies in the definition of **custom attributes** within the PREEvision modeling environment. Although PREEvision already offers an extensive meta-model specific to the automotive domain, suited to supporting the modeling of complex E/E architectures, there are cases where the predefined attributes are not sufficient to represent specific architectural properties or desired dynamic behaviors. In such situations, PREEvision allows the meta-model to be extended by creating custom attributes, which behave like native attributes and can be applied at any level of abstraction, from software components to hardware elements.


SW_Attributes

AlternativeRoute SW Type (Software Architecture Diagram)

ServiceAttributes

Interruptible	✓
Relocatable	✓
Priority	2

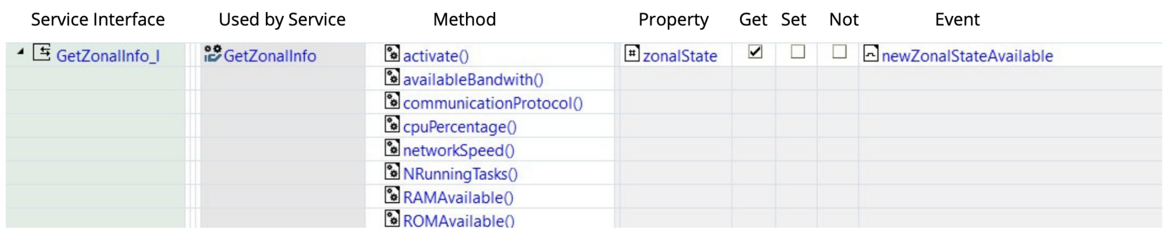
Figure 5.9: Custom attributes

In the context of this thesis, three custom attributes associated with each modeled software service have been defined and are shown in Figure 5.9.

- **Priority** (integer): indicates the criticality level of the service with respect to the overall functioning of the system.
- **Relocatable** (boolean): specifies whether the service can be dynamically migrated from one ECU to another during run-time.
- **Interruptible** (boolean): defines whether the service can be temporarily suspended or interrupted in case of resource lack.

These attributes were introduced with the aim of supporting and testing the concept of **dynamic service shifting**, i.e., the ability to reallocate software services in real time between ECUs according to their operating conditions. This approach represents an evolution from traditional static deployment models, in which each service is permanently assigned to a specific ECU, without taking into account the actual computational load or resource availability during execution.

In combination with custom attributes, the *Get Zonal Info* service has been defined in PREEvision modeling, as shown in Figure 5.10. This service has been designed to enable the dynamic exchange of operational information between ECUs, such as current CPU load, RAM and ROM availability, or the number of running tasks.



Service Interface	Used by Service	Method	Property	Get	Set	Not	Event
GetZonalInfo_I	GetZonalInfo	activate()	zonalState	✓	☐	☐	newZonalStateAvailable
		availableBandwith()					
		communicationProtocol()					
		cpuPercentage()					
		networkSpeed()					
		NRunningTasks()					
		RAMAvailable()					
		ROMAvailable()					

Figure 5.10: Get zonal info service

The main goal is to provide the system with an up-to-date overview of the computational load and any malfunctions of each ECU, so as to enable real-time assessment of the advisability of transferring certain software services. If a service is marked as "*relocatable*" and "*interruptible*", and another ECU has sufficient spare capacity, the system can decide to migrate the service safely. The priority attribute further helps establish a hierarchy among services, which is useful for deciding which ones should be kept active, suspended, or reallocated in critical situations, such as overloads, partial failures, or system degradation.

This flexible service shifting strategy opens the way for more resilient and adaptive software architectures, in which runtime conditions determine the **allocation of services dynamically**.

5.6 Hardware architecture

After defining and modeling the software architecture according to both paradigms, we moved on to designing the hardware architecture, following a **zonal model**.

In particular, the vehicle was logically divided into four main zones: Front Left, Front Right, Rear Left, and Rear Right. Each zone is coordinated by a **Zonal Controller**, responsible for managing local control units and communicating with other controllers in neighboring zones, if necessary. This configuration reflects one of the emerging approaches in the automotive industry to reduce wiring complexity, improve internal communication efficiency, and simplify system maintenance.

The ECUs have been placed in a specific zone (area) of the vehicle due to their physical and functional proximity. These ECUs are connected via a shared **local**

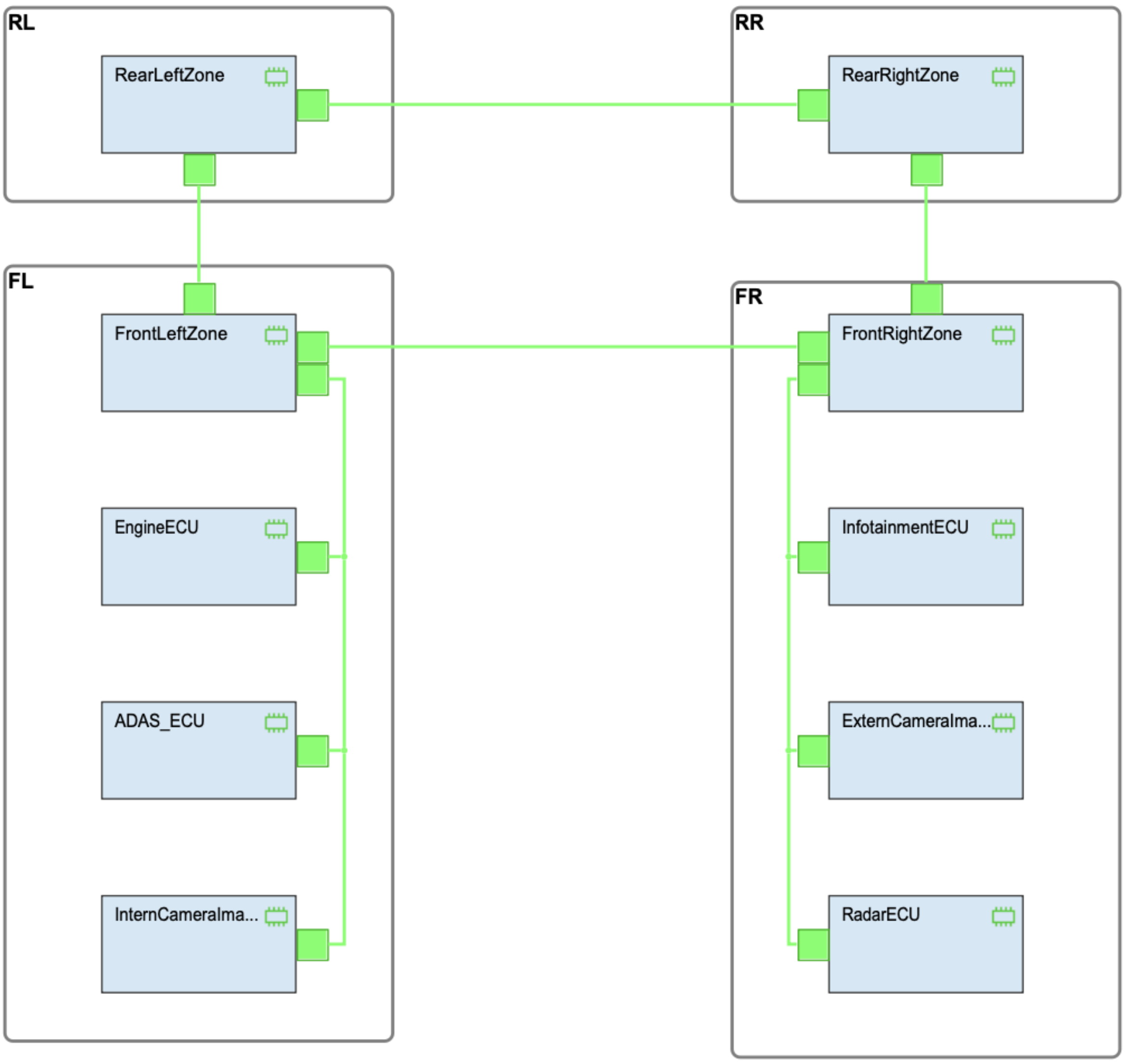


Figure 5.11: Hardware Architecture

Ethernet network (LAN), managed by the respective zonal controller.

Communication between ECUs belonging to different zones does not take place directly, but the only point of interconnection between the zones is represented by the respective zonal controllers, which act as **intelligent gateways** for inter-zone traffic.

Figure 5.11 shows the modeling of the zonal architecture implemented in PREEvision. It can be seen that **each block represents an ECU**, while the lines connecting the blocks indicate the **Ethernet connections** modeled in the system. These connections represent the logical topology of the network, which reflects the physical configuration planned for a vehicle with a zonal architecture.

In particular, **each zone is equipped with a local Ethernet backbone**, to which all ECUs in the zone are connected. This diagram simulates a **vehicle LAN**, in which messages exchanged between ECUs are carried over the common bus, enabling direct and low-latency communication within the zone. Every message sent over that Ethernet link is potentially visible to all ECUs in the zone, optimizing the required bandwidth.

The connections between the different zonal controllers, visible in the central or upper part of the figure, constitute the **inter-zone backbone network**, which is necessary to enable communication between different zones.

In the context of modeling, these connections are also essential for the subsequent **configuration of SOME/IP instances**, which will be transmitted over these Ethernet lines according to the parameters defined in the previous steps.

In this topology, for simplicity, the rear zones contain only their respective zone controllers, without additional local ECUs. This simplification is due to the fact that the aim of the project is not to completely model the architecture of an entire vehicle as said before.

In the front zones, on the other hand, the control units that could realistically host the services defined in the thesis project, have been positioned. In particular, the **Front Left zone** contains:

- *Engine ECU*
- *ADAS ECU*
- *Intern Camera Image ECU*

This location reflects both the physical proximity typical of the actual vehicle and the functional logic, as these nodes are expected to participate in the generation or consumption of services related, for example, to driving behavior analysis or driving hours management.

The **Front Right zone** instead includes:

- *Infotainment ECU*
- *Extern Camera Image ECU*
- *Radar ECU*

Again, the layout has been designed taking into account the physical location of the control units in the actual vehicle and the need to correctly model communications between providers and consumers of the defined services, such as traffic analysis or the generation of an alternative route.

5.6.1 Mapping software/hardware layers

After modeling the software and hardware architectures in PREEvision, the **software-hardware mapping** phase was carried out. Mappings are model artifacts that connect artifacts from different levels of the E/E architecture. The mapping between one architecture level and another represents a consistent relationship between the components of the E/E architecture.

This operation is necessary to assign each software component to a specific ECU within the defined topology, enabling the automatic generation of signals and communication channels at the network level, as provided by PREEvision.

In the case of **software architecture** inspired by the Classic paradigm, each *Application Software Component* has been mapped to an ECU within its corresponding zones. In this model, components communicate with each other via *Sender/Receiver ports*, connected via *Data Elements* and *Signals.*

So, after mapping, the automatic *Signal Routing* algorithm provided by PREEvision can generate the relevant *System Signals* and *PDUs* (see section below), preparing the system for communication on Ethernet or CAN buses (not explicitly modeled in this thesis, as discussed in section 5.4).

In **service-oriented software architecture**, on the other hand, mapping has a more flexible strategy: what is assigned to the ECUs are not the application components, but the service instances they provide or consume. Thanks to support for the **AUTOSAR Adaptive** standard, it has been possible to define client and server instances for each service and logically place them on the ECUs through mapping.

In this case, communication is no longer managed via *Signal* and *Data Elements*, but via a *Service Interface* configured at the network level through the SOME/IP middleware functionality. Each service instance is associated with an IP address, a port, and a transport protocol (TCP or UDP).

In both cases, PREEvision allowed the mapping to be visualized within the zone topology, as we can see in Figures 5.12 and 5.13.

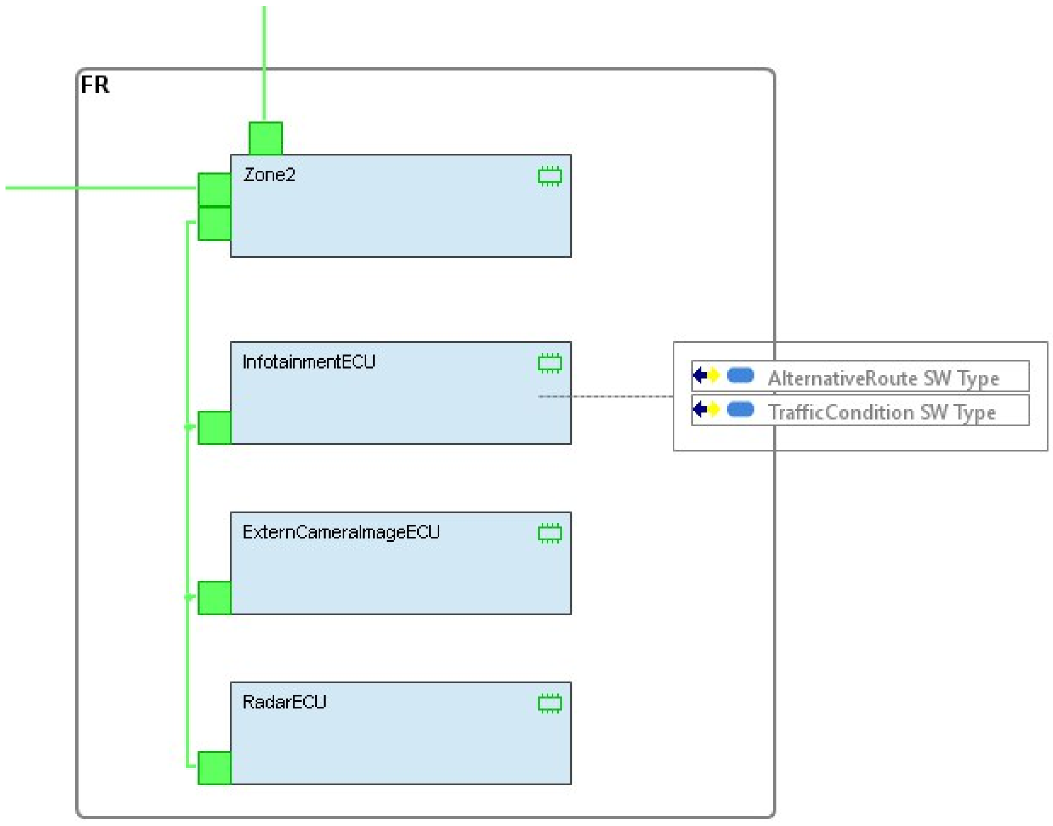


Figure 5.12: Mapping software architecture

This two figures illustrate the **mapping** on the *Infotainment ECU* located in the **Front Right Zone**.

Figure 5.12 shows the mapping created according to **software-oriented architecture**. In this case, the *Application Software Components* (Alternative Route SW Type and Traffic Condition SW Type) responsible for the internal logic of the *Traffic Condition* and *Alternative Route* services have been mapped directly onto the same ECU. These components, implemented according to the AUTOSAR Classic style, directly contain the application functionality and should be connected to sensors and actuators via S/R ports.

Figure 5.13 shows the mapping according to the **service-oriented approach**: the provider instances of the previously mentioned services, such as *Traffic Condition Provider* and *Alternative Route Provider*, are mapped to the *Infotainment ECU* in accordance with the AUTOSAR Adaptive paradigm. In this context, what is assigned to the ECU is not a monolithic application block, but rather the functional role (provider or consumer) associated with each service.

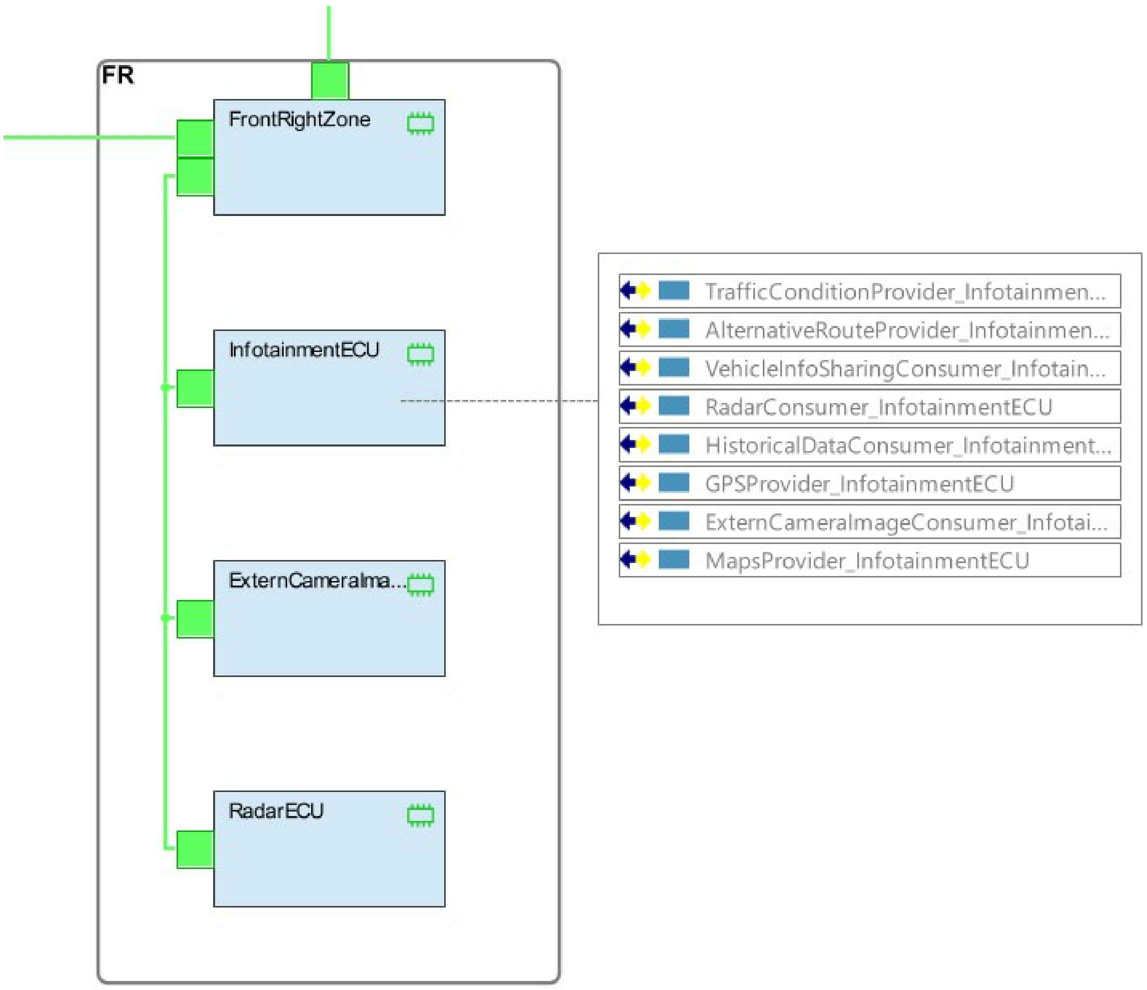

FR
FrontRightZone
InfotainmentECU
ExternCameraIma...
RadarECU
TrafficConditionProvider_Infotainmen...
AlternativeRouteProvider_Infotainmen...
VehicleInfoSharingConsumer_Infotain...
RadarConsumer_InfotainmentECU
HistoricalDataConsumer_Infotainment...
GPSProvider_InfotainmentECU
ExternCameraImageConsumer_Infotai...
MapsProvider_InfotainmentECU

Figure 5.13: Mapping service architecture

5.7 Communication layer

Once the services and software interfaces, as well as the hardware architecture of the system, have been modeled, it is possible to proceed with the modeling of the **communication layer**, which is the layer responsible for data transmission within the vehicle. This step is important to ensure consistency and traceability between the functional model and the implementation of communication on the Ethernet network.

In this project, the communication layer was modeled exclusively for the service architecture, as this represents the main focus of the thesis work.

At the heart of this phase is the **Signal Routing** module, an advanced tool provided by PREEvision to automate the generation and configuration of communication artifacts. The signal router is designed to determine the best transmission paths through a predefined switched Ethernet network topology, taking into account the presence of VLANs and existing network configurations. In mixed or legacy systems, it is also able to suggest the optimal points for inserting inter-bus gateways, for example between CAN and Ethernet, thus improving communication efficiency and automating failure-prone design steps.

During this phase, the **software interfaces** – whether **Sender/Receiver** or **Client/Server** – are directly associated with the communication elements. In the first case, the **Data Elements** become the data transmitted from one component to another; in the second case, **Operations** represent methods that can be invoked, synchronously or asynchronously, according to the client/server paradigm.

The PREEvision *Signal Router* is not only capable of automatically generating Signals, but also of creating the necessary **System Signal Mappings**. This makes associations between *Data Elements/Operations* and the respective Signals, i.e., the entities that represent the data actually transferred through the vehicle's physical network.

In addition, the *Signal Router* also handles the physical instantiation of signals on the **bus**, i.e., the creation of **Signal Transmissions**.

Therefore, among the communication artifacts necessary to complete the design of Ethernet network communication, we find:

- **Signals**: representing the instances of transmitted data.
- **PDUs (Protocol Data Units)**: logical containers for signals.
- **Signal Transmissions and PDU Transmissions**: which define the methods, timing, and priorities with which signals and PDUs should be sent over the bus in the network.
- **Socket Addresses, Application Endpoints, and Network Endpoints**: uniquely identify the sender and recipient at the network level.
- **Consumed Service Instances and Provided Service Instances**: specify which services are requested and offered by each ECU.

All these elements are automatically generated and integrated into the model, reducing the possibility of human error in the most complex phases of design. Furthermore, thanks to tight integration with AUTOSAR standards, the generated artifacts are consistent with the ARXML structure and can be exported for final implementation.

Figure 5.14 shows the ECU involved in the **communication generated from a specific signal**. In particular, the signal shown is associated with the response of the `get_route()` method belonging to the *Alternative Route Service* (not visible in the figure for reasons of clarity). In this context, the *Infotainment ECU*, highlighted in orange, acts as the **sender**, i.e., the component that provides the response to the method. The *Front Right Zone* control unit, highlighted in yellow, acts as the **receiver**, i.e., the recipient of the response. This example clearly shows

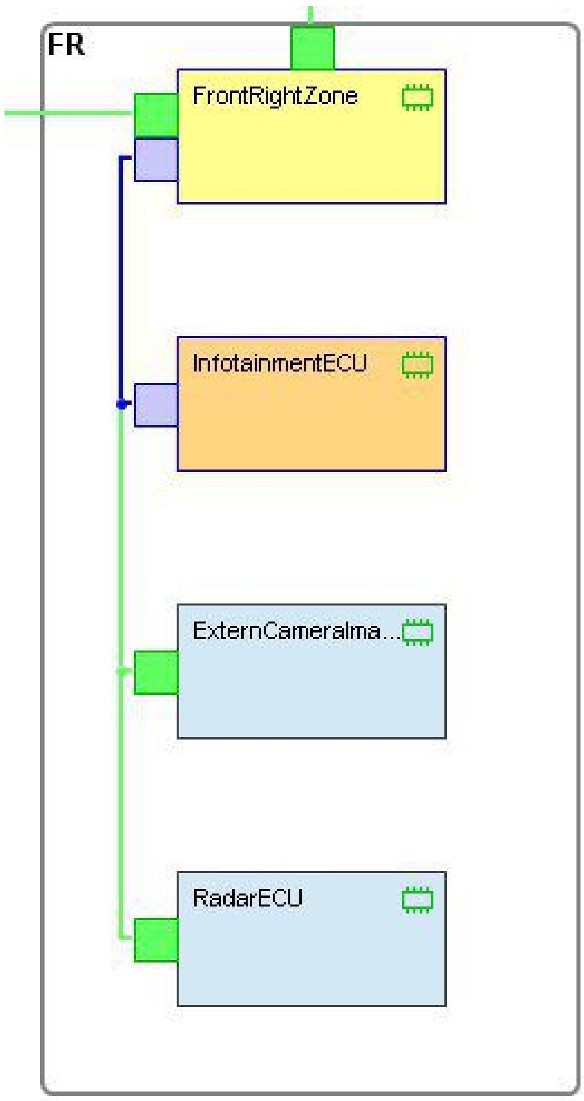


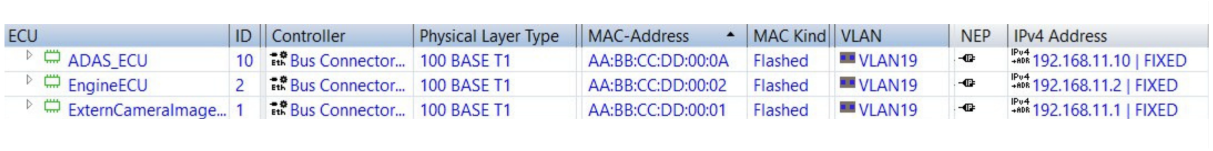
Figure 5.14: Signal mapping between infotainment and front right zone ECU

how, within a service-oriented architecture, it is possible to explicitly model the flow of information between the various ECUs involved in the system.

In addition to Signal Routing, PREEvision provides a series of tables that allow to define the details of the **physical and datalink layer** (OSI Layer 1 and 2). In particular, it is possible to configure **Ethernet controllers** by specifying their **MAC** addresses and the type of **physical connection** (e.g., 100BASE-T1). In addition, **Network Endpoints** can be defined, i.e., the network nodes at **OSI Layer 3**, to which the **IP addresses**, required for communication between ECUs in an Ethernet environment, can be assigned.

Figure 5.15 shows details of some of the nodes involved in Ethernet communication. In particular, it shows the *ADAS*, *Engine*, and *External Camera Image* control units, each of which is equipped with an Ethernet controller (represented as **Bus Connector**).

All control units use a **100BASE-T1** physical interface, a standard designed



ECU	ID	Controller	Physical Layer Type	MAC-Address	MAC Kind	VLAN	NEP	IPv4 Address
ADAS_ECU	10	Bus Connector...	100 BASE T1	AA:BB:CC:DD:00:0A	Flashed	VLAN19		192.168.11.10 | FIXED
EngineECU	2	Bus Connector...	100 BASE T1	AA:BB:CC:DD:00:02	Flashed	VLAN19		192.168.11.2 | FIXED
ExternCameraImage...	1	Bus Connector...	100 BASE T1	AA:BB:CC:DD:00:01	Flashed	VLAN19		192.168.11.1 | FIXED

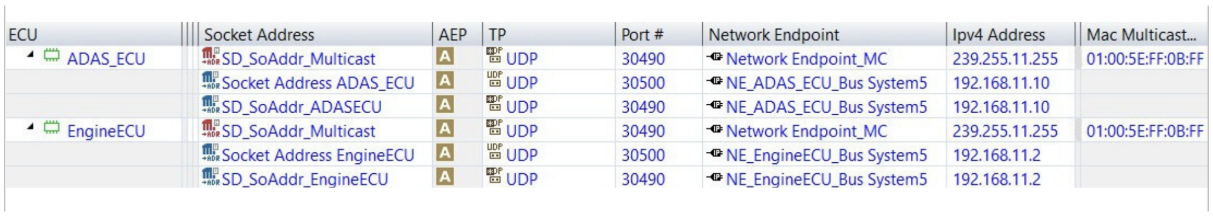
Figure 5.15: ECU Ethernet configuration

specifically for the automotive sector. This technology allows data transmission at **100 Mbps** over a single twisted pair, ensuring robustness, compatibility with critical electromagnetic environments, and a significant reduction in weight and wiring complexity compared to traditional Ethernet solutions (e.g., 100BASE-TX). The choice of 100BASE-T1 is therefore ideal for modern automotive Ethernet-based architectures.

The following is also defined for each controller:

- **The MAC address**, unique for each network device;
- **The MAC address assignment mode**, set in this case as Flashed (i.e., preconfigured and written to the device memory);
- **The VLAN to which it belongs**, which in this configuration is common to all ECUs shown, simplifying the broadcast domain;
- **The Network Endpoint**, to which a static IPv4 address is associated. It is also possible to configure IPv6 addresses, but this has not been done in this project.

Next, it is possible to generate automatically **sockets** (OSI Layer 4) for each node in the network. Each socket address consists of an IP/port pair and represents an *Application Communication Endpoint*, which is essential for establishing connections between services distributed over an Ethernet network.



ECU	Socket Address	AEP	TP	Port #	Network Endpoint	Ipv4 Address	Mac Multicast...
ADAS_ECU	SD_SoAddr_Multicast	A	UDP	30490	Network Endpoint_MC	239.255.11.255	01:00:5E:FF:0B:FF
	Socket Address ADAS_ECU	A	UDP	30500	NE_ADAS_ECU_Bus System5	192.168.11.10	
	SD_SoAddr_ADASECU	A	UDP	30490	NE_ADAS_ECU_Bus System5	192.168.11.10	
EngineECU	SD_SoAddr_Multicast	A	UDP	30490	Network Endpoint_MC	239.255.11.255	01:00:5E:FF:0B:FF
	Socket Address EngineECU	A	UDP	30500	NE_EngineECU_Bus System5	192.168.11.2	
	SD_SoAddr_EngineECU	A	UDP	30490	NE_EngineECU_Bus System5	192.168.11.2	

Figure 5.16: Socket configuration in the Ethernet network

Figure 5.16 shows the ADAS and *Engine* control units again, which are associated with different socket addresses. In particular, each ECU has three socket addresses:

- **a multicast socket**, dedicated to the Service Discovery (SD) mechanism provided by the SOME/IP protocol, to announce the presence of services on the network.

- **a unicast socket**, used for point-to-point communication (unicasting) of service-related data.

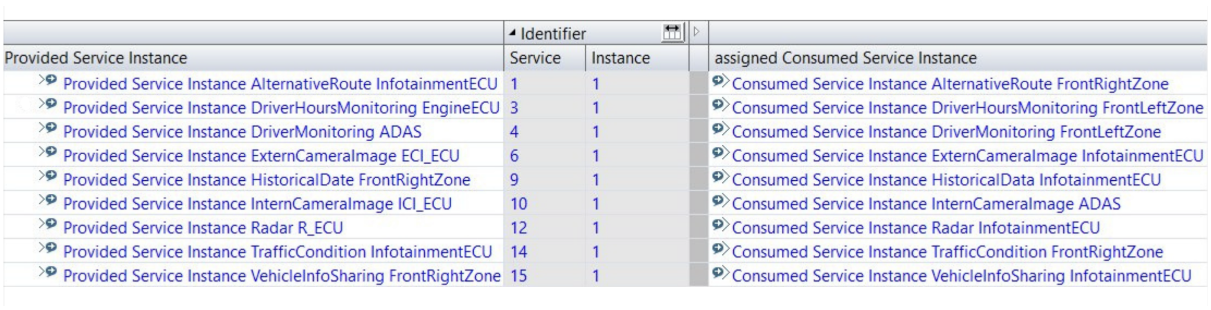
- **an additional unicast socket**, reserved for sending Service Discovery responses.

All sockets use **UDP** as their transport protocol and have a corresponding **port number**, specified in the relevant *Application Endpoint*, while the *Network Endpoint* assigns the IP address, which in all cases coincides with the one already seen in the previous Ethernet configuration. For multicast sockets, the corresponding **MAC multicast** address is also indicated, which is used to receive service alerts on the network.

The presence of two unicast sockets with the same IP address may initially appear redundant, but each socket is uniquely identified by the IP, port, and protocol combination, and the separation between sockets intended for service data and those intended for discovery management allows for a more **modular** and **robust** system organization.

This configuration, in particular, is in line with the specifications of **SOME/IP-SD**, which supports a **distributed discovery** infrastructure based on periodic multicast messages and punctual unicast responses.

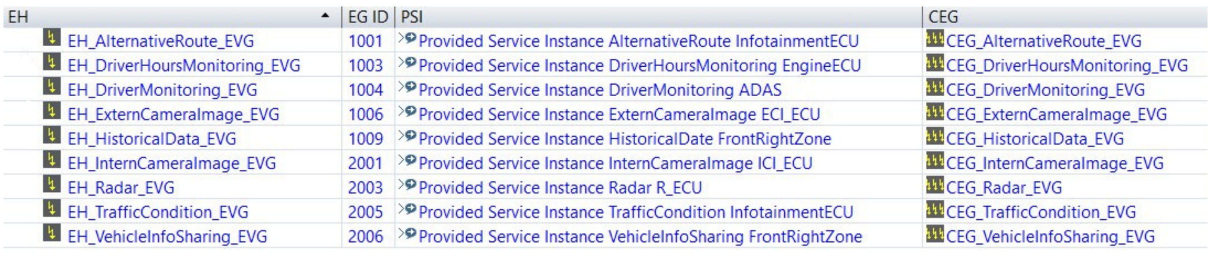
In this phase, the links between the **Provided Service Instances** and the **Consumed Service Instances** are also defined, i.e., respectively, the service instances offered by an ECU and those expected by another ECU, as shown in Figure 5.17. This association is necessary to establish correct **communication routing** in the context of the service-oriented paradigm.



	Identifier		
Provided Service Instance	Service	Instance	assigned Consumed Service Instance
Provided Service Instance AlternativeRoute InfotainmentECU	1	1	Consumed Service Instance AlternativeRoute FrontRightZone
Provided Service Instance DriverHoursMonitoring EngineECU	3	1	Consumed Service Instance DriverHoursMonitoring FrontLeftZone
Provided Service Instance DriverMonitoring ADAS	4	1	Consumed Service Instance DriverMonitoring FrontLeftZone
Provided Service Instance ExternCameraImage ECI_ECU	6	1	Consumed Service Instance ExternCameraImage InfotainmentECU
Provided Service Instance HistoricalDate FrontRightZone	9	1	Consumed Service Instance HistoricalData InfotainmentECU
Provided Service Instance InternCameraImage ICI_ECU	10	1	Consumed Service Instance InternCameraImage ADAS
Provided Service Instance Radar R_ECU	12	1	Consumed Service Instance Radar InfotainmentECU
Provided Service Instance TrafficCondition InfotainmentECU	14	1	Consumed Service Instance TrafficCondition FrontRightZone
Provided Service Instance VehicleInfoSharing FrontRightZone	15	1	Consumed Service Instance VehicleInfoSharing InfotainmentECU

Figure 5.17: Mapping between service instances provided and consumed

Each service can also have one or more **Event Groups** associated with it, which merge corresponding events to be transmitted as logical units. Associating *Event Groups* with *Service Instances* allows for efficient management of asynchronous communication, typical of SOME/IP protocol.



EH	EG ID	PSI	CEG
EH_AlternativeRoute_EVG	1001	Provided Service Instance AlternativeRoute InfotainmentECU	CEG_AlternativeRoute_EVG
EH_DriverHoursMonitoring_EVG	1003	Provided Service Instance DriverHoursMonitoring EngineECU	CEG_DriverHoursMonitoring_EVG
EH_DriverMonitoring_EVG	1004	Provided Service Instance DriverMonitoring ADAS	CEG_DriverMonitoring_EVG
EH_ExternCameraImage_EVG	1006	Provided Service Instance ExternCameraImage ECI_ECU	CEG_ExternCameraImage_EVG
EH_HistoricalData_EVG	1009	Provided Service Instance HistoricalDate FrontRightZone	CEG_HistoricalData_EVG
EH_InternCameraImage_EVG	2001	Provided Service Instance InternCameraImage ICI_ECU	CEG_InternCameraImage_EVG
EH_Radar_EVG	2003	Provided Service Instance Radar R_ECU	CEG_Radar_EVG
EH_TrafficCondition_EVG	2005	Provided Service Instance TrafficCondition InfotainmentECU	CEG_TrafficCondition_EVG
EH_VehicleInfoSharing_EVG	2006	Provided Service Instance VehicleInfoSharing FrontRightZone	CEG_VehicleInfoSharing_EVG

Figure 5.18: Mapping of event groups between providers and consumers

In Figure 5.18, for each *Event Group*, the relative ID and association with the corresponding *Provided Service Instance* are visible, i.e., the control unit responsible for generating and transmitting the events belonging to that group. Each *Event Group* is also associated with the respective *Consumed Event Group*, which represents the logical link with the service consumers who subscribe to the events emitted. This link allows updates to be propagated in **Publish/Subscribe** mode according to the SOA paradigm provided by the SOME/IP protocol.

Finally, to enable the correct functioning of **Service Discovery**, PREEvision automatically generates all the **multicast communication artifacts** required for ECUs belonging to the same *Ethernet Cluster* to dynamically discover the services available on the network. These artifacts include the configuration of ports and multicast addresses, as required by the SOME/IP-SD protocol.

Once the modeling of all architectural levels has been completed, it is possible to proceed with exporting the *ARXML* file, which includes all the specifications defined in this chapter. In particular, the file will contain information relating to the modeling of the *service-oriented* architecture, which will create the basis for the practical implementation described in the next chapter.

Chapter 6

Building C++ application using SOME/IP

After modeling the software and hardware architecture in PREEvision and exporting the **ARXML** file, the second phase of the project started, which was dedicated to the actual implementation by creating a **C++ application**. The goal of this phase is to simulate the behavior of two nodes, one acting as a provider and one acting as a consumer, located on two different control units using the **vSomeIP** middleware.

So this chapter analyzes the implementation phase of the architecture previously modeled in PREEvision.

The workflow can be summarized in three main steps, also illustrated in Figure 5.1 in Chapter 5:

- **Parsing the ARXML file**: a *Python* parser processing the PREEvision output to extract relevant information about SOME/IP services and generate a *JSON* file that is simplified and more easily interpretable.
- **Configuration file generation**: the extracted data is used to dynamically create the `"provider.json"` and `"consumer.json"` files required by vSomeIP for the correct startup of participants on the Ethernet network, with a C++ executable.
- **Development of C++ applications**: two separate applications have been developed, one to simulate the behavior of a **provider** and one to simulate a **consumer**, in accordance with the architectural model.

This process, excluding modeling on PREEvision, can be automated through a **custom bash** file `"setup.sh"`, which runs these steps sequentially.

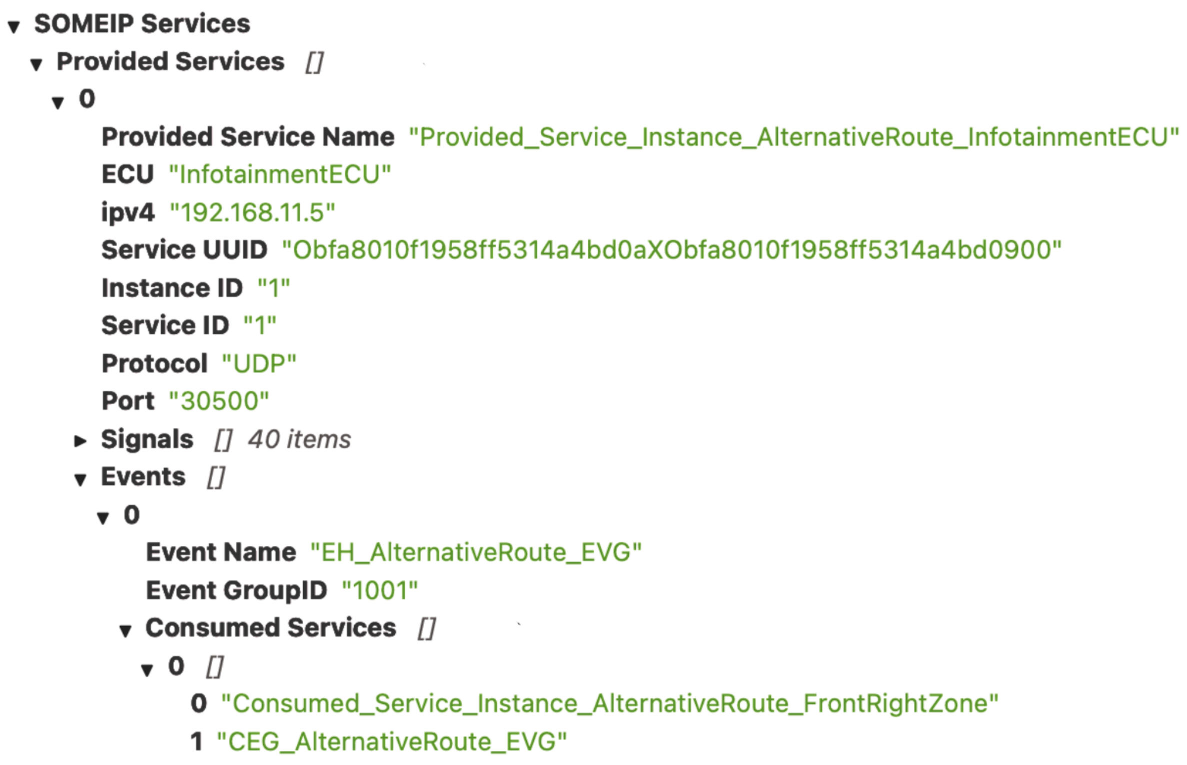
The following paragraphs describe in detail the technical choices adopted, the contents of the generated *JSON* file, the structure of the vSomeIP configuration files, and the logic for implementing the two simulated nodes.

6.1 Parsing the ARXML file and generating the JSON file

To process the **ARXML** file exported from PREEvision, a custom parser was developed in **Python**. Although there are modules dedicated to reading ARXML files, *Python* was found to be a more direct and faster approach using the *lxml* library to read and navigate the XML tree. Developing a custom script also made it possible to understand the structure of the file itself.

The goal was not to cover the entire spectrum of information contained in *ARXML*, but only to extract the minimum information necessary for implementing the *C++* application, including:

- All SOME/IP services (providers and consumers);
- All Service Interfaces associated with the above services;
- SOME/IP Deployments, i.e., all IDs associated with SOME/IP elements;
- Information to enable multicast traffic useful for Service Discovery;



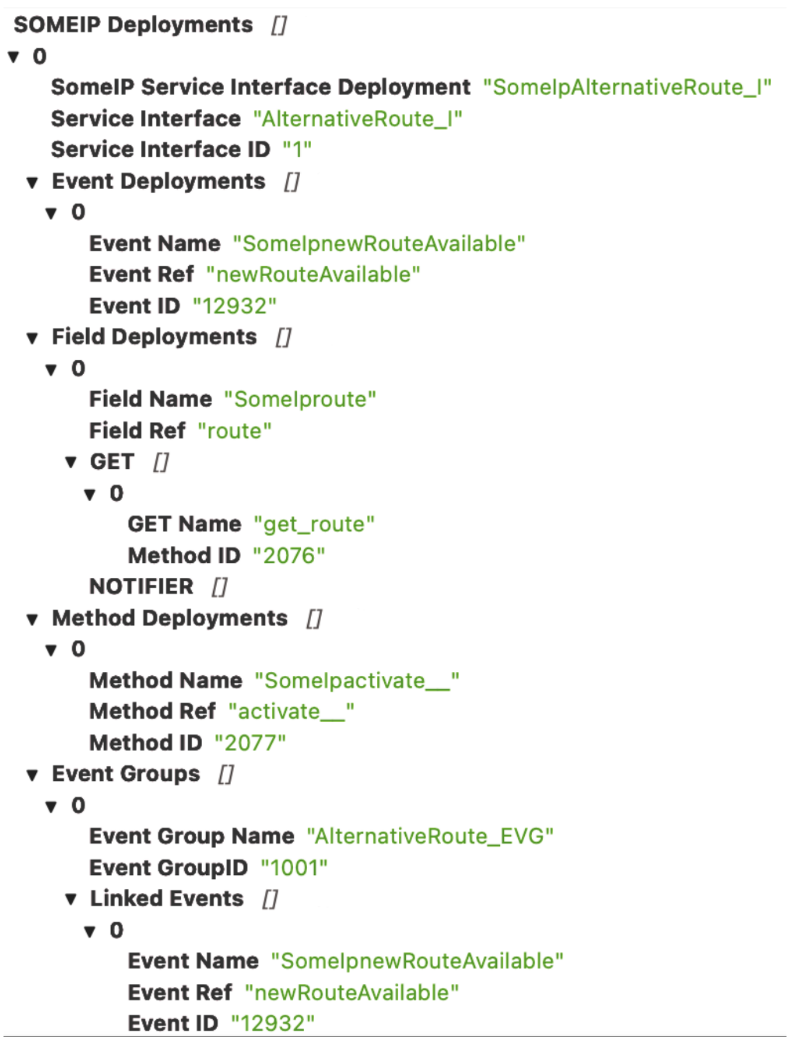
```
▼ SOMEIP Services
  ▼ Provided Services  []
    ▼ 0
        Provided Service Name  "Provided_Service_Instance_AlternativeRoute_InfotainmentECU"
        ECU  "InfotainmentECU"
        ipv4  "192.168.11.5"
        Service UUID  "Obfa8010f1958ff5314a4bd0aXObfa8010f1958ff5314a4bd0900"
        Instance ID  "1"
        Service ID  "1"
        Protocol  "UDP"
        Port  "30500"
      ▶ Signals  []  40 items
      ▼ Events  []
        ▼ 0
            Event Name  "EH_AlternativeRoute_EVG"
            Event GroupID  "1001"
          ▼ Consumed Services  []
            ▼ 0  []
                0  "Consumed_Service_Instance_AlternativeRoute_FrontRightZone"
                1  "CEG_AlternativeRoute_EVG"
```

Figure 6.1: SOME/IP services

For each service modeled in PREEvision, the parser extracted the relevant information and structured it in *JSON* format, as shown in Figure 6.1. The image shows, for example, the data associated with the **Alternative Route Service**, already seen in the PREEvision modeling discussed in the previous chapter.

In particular, the following information is listed for each service:

- the **provider instance**, mapped to the *Infotainment ECU*;
- the **IPv4 address** on which the service will be available;
- the **UUID**, the **service ID** and the **instance ID**, which uniquely identify the service in the SOME/IP context;
- the **communication protocol** (in this case UDP) and the corresponding **port**;
- the **event offered** by the service and the corresponding **event group ID**;
- the list of **signals** transmitted that are associated with the service;
- the **consumer instance** of the service and the control unit on which it is mapped, in this case the *Zonal Controller Front Right*.



```
SOMEIP Deployments  []
▼ 0
    SomeIP Service Interface Deployment  "SomeIpAlternativeRoute_I"
    Service Interface  "AlternativeRoute_I"
    Service Interface ID  "1"
  ▼ Event Deployments  []
    ▼ 0
        Event Name  "SomeIpnewRouteAvailable"
        Event Ref  "newRouteAvailable"
        Event ID  "12932"
  ▼ Field Deployments  []
    ▼ 0
        Field Name  "SomeIproute"
        Field Ref  "route"
      ▼ GET  []
        ▼ 0
            GET Name  "get_route"
            Method ID  "2076"
        NOTIFIER  []
  ▼ Method Deployments  []
    ▼ 0
        Method Name  "SomeIpactivate__"
        Method Ref  "activate__"
        Method ID  "2077"
  ▼ Event Groups  []
    ▼ 0
        Event Group Name  "AlternativeRoute_EVG"
        Event GroupID  "1001"
      ▼ Linked Events  []
        ▼ 0
            Event Name  "SomeIpnewRouteAvailable"
            Event Ref  "newRouteAvailable"
            Event ID  "12932"
```

Figure 6.2: Service deployments

Each modeled service is associated with an interface, consisting of methods, events, and fields. For each of these interfaces, all the identifiers necessary to enable communication according to the SOME/IP protocol on an Ethernet network have been generated. Figure 6.2 shows the **SOME/IP Deploy** relating to the *Alternative Route Service* interface.

The **Service Interface ID** is visible, a unique identifier for each service, which corresponds to the **Service ID** already presented in the previous figure. This link ensures consistency between the two blocks of the *JSON* file and allows consistent reading of the available data.

In addition, all **events** associated with the **interface** are listed, with their respective names (both the name of the SOME/IP signal and the name of the event to which they refer) and their **Event IDs**. Any **fields** with possible associated operations are then listed: *get, set, notifier,* each with its own name and unique identifier.

This is followed by the **methods**, also defined by name and **Method ID**. Finally, the **Event Group** is indicated, which collects one or more events, specifying the **Event Group ID** and the list of events that are part of it.

The JSON format was chosen because it is the de facto standard for data exchange between applications, ensuring portability and simplifying any future migration to a different programming language.

This representation allows clear mapping between the logical modeling performed in PREEvision and the subsequent *C++* implementation based on vSomeIP.

6.2 Generating configuration files for vSomeIP

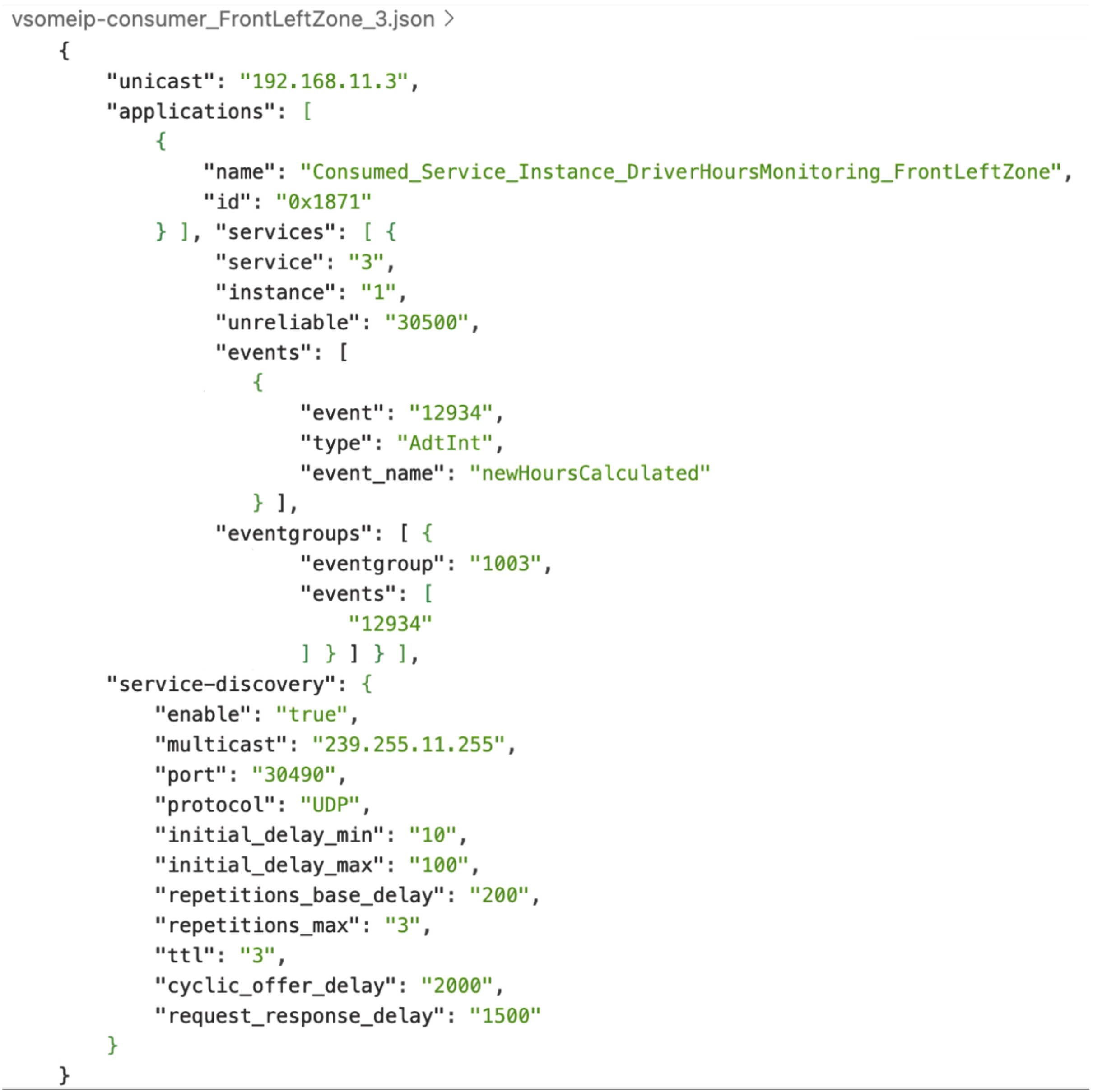
After parsing the *ARXML* with the *Python* tool, the next step is to generate the configuration files, which are again in *JSON format*, used by vSomeIP. These files are essential for the correct initialization of the **SOME/IP middleware** and contain:

- **Unique IP network address**
- **Applications and client IDs**: each participant (server or client) has an applications section with a unique name and ID;
- **Services and instances**: services section, containing *service ID*, *instance ID*, and associated TCP or UDP port (configured as "*reliable*" or "*unreliable*", respectively). Depending on if it's the case of providers or consumers, the services will be those provided or consumed.
- **Events**: in the services section, there are also events offered/requested by the instance with ID, name, type, and associated Event Group.

- **Service discovery**: configurations related to multicast, port, protocol, and timing used for dynamic service discovery.

An example of a configuration file is shown in Figure 6.3.

Specifically, the consumer instance of the *Driver Hours Monitoring service* mapped to the *Front Left control unit* is shown.



```
vsomeip-consumer_FrontLeftZone_3.json >
    {
        "unicast": "192.168.11.3",
        "applications": [
            {
                "name": "Consumed_Service_Instance_DriverHoursMonitoring_FrontLeftZone",
                "id": "0x1871"
            } ], "services": [ {
                 "service": "3",
                 "instance": "1",
                 "unreliable": "30500",
                 "events": [
                     {
                         "event": "12934",
                         "type": "AdtInt",
                         "event_name": "newHoursCalculated"
                     } ],
                 "eventgroups": [ {
                         "eventgroup": "1003",
                         "events": [
                             "12934"
                         ] } ] } ],
        "service-discovery": {
            "enable": "true",
            "multicast": "239.255.11.255",
            "port": "30490",
            "protocol": "UDP",
            "initial_delay_min": "10",
            "initial_delay_max": "100",
            "repetitions_base_delay": "200",
            "repetitions_max": "3",
            "ttl": "3",
            "cyclic_offer_delay": "2000",
            "request_response_delay": "1500"
        }
    }
```

Figure 6.3: Configuration file

Generation occurs as follows:

1. The *Python* script reads the *ARXML* file correctly, parses it, and creates a *JSON* file, extracting all relevant metadata. In a *C++* module, this data is used to produce the vSomeIP configuration files: for example, "*vsomeip-consumer_FrontLeftZone.json*" will contain the network parameters, client ID, and list of services consumed by the *Front Left Zone control unit.*

2. These configuration files, in *JSON* format, are then read at launching by the *C++* application integrated with vSomeIP, which we will discuss in the next section.

The advantage is dual:

- The MBSE model is kept synchronized with the actual middleware, also eliminating possible manual errors.
- The approach is modular: it is possibile to have separate configurations for each ECU (provider or consumer), which can be easily generated automatically.

6.3 Implementation of the C++ application with vSomeIP

Once the configuration files were generated in *JSON* format, the implementation of the **C++ applications** was started to simulate the behavior of the control units modeled in PREEvision. In particular, for each ECU involved in the communication of the defined services (e.g., *Alternative Route, Traffic Condition*), a dedicated configuration file was created containing information on the services offered or consumed by that specific ECU.

Starting from these files, two separate *C++* programs were developed, each representing a control unit of interest:

- one for the control unit acting as the **service provider**;
- one for the control unit acting as the **service consumer**.

For communication, the **vSomeIP middleware** was adopted, and the applications are based on the use of the API it provides, following the classic asynchronous structure:

- The provider registers a service by specifying a *service ID* and an *instance ID*, opens a socket on a TCP or UDP port, and waits for connections or notifications.
- The consumer uses service discovery functions to locate the service offered and, once found, sends requests (method calls) or subscribes to asynchronous events (event subscription).

Both applications read the *JSO*N file corresponding to their configuration, and the environment is initialized using the "$runtime :: get() - > init()$" function.

The interaction between the two simulated control units confirmed **correct alignment**: services are published and consumed according to the configuration derived directly from PREEvision, ensuring perfect consistency between the modeling phase and the prototype implementation.

How vSomeIP works As shown in Figure 6.4, vSomeIP manages both SOME/IP interactions between devices (external communication) and those between local processes on the same ECU.

Two devices communicate via so-called communication endpoints, which determine the transport protocol used (TCP or UDP) and its parameters, such as the port number or other parameters. All these parameters are configuration parameters that are set in the vSomeIP configuration files (described in the previous section).

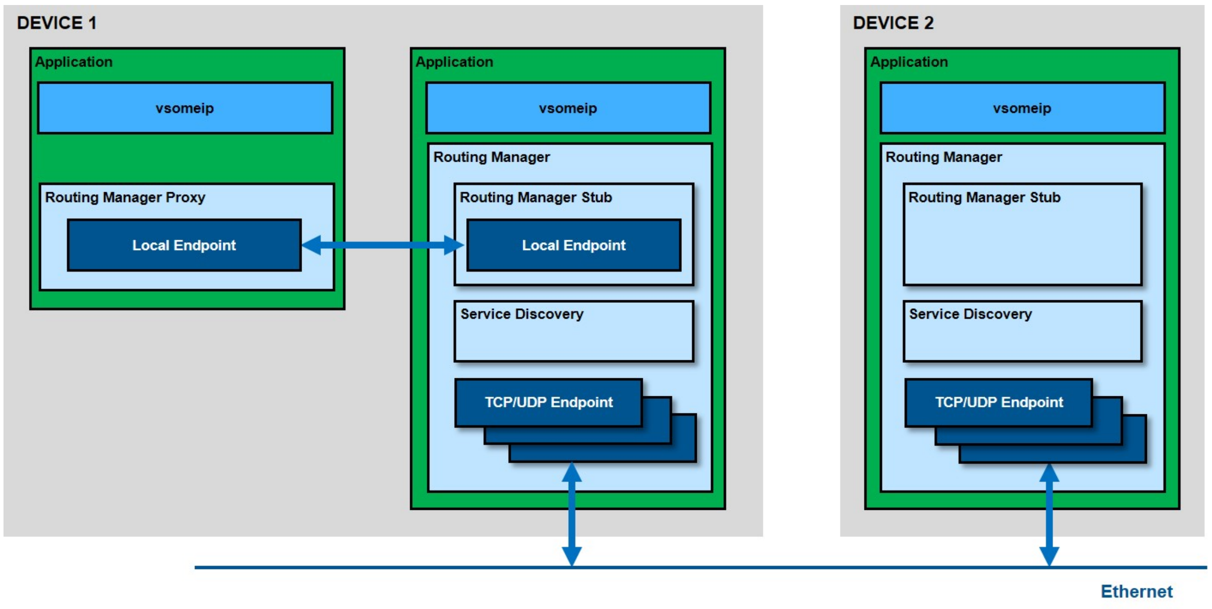


Figure 6.4: vSomeIP overview

The central element of the middleware is the **Routing Manager**, an internal process that manages message routing between providers and consumers, both within the same ECU (local endpoints) and between different ECUs. Internal communication takes place via local endpoints implemented by UNIX domain sockets. Since this internal communication is not routed through a central component (such as the D-Bus daemon), it is very fast.

In each node, there is only one Routing Manager per device; if nothing is configured, the first vsomeip application running that calls "$vsomeip :: runtime :: get() - > init()$" also starts the Routing Manager. The other subsequent ones behave as local clients (proxy) that interface with the Routing Manager via UNIX sockets. If, on the other hand, the applications are distributed on different machines or ECUs, each one runs its own Routing Manager, and communication happens via the configured Ethernet network, according to the rules established in the *JSON* configuration files.

Service Discovery is also managed by the Routing Manager, using the SOME/IP-SD protocol. Each provider announces the availability of its services, indicating the *service ID, instance ID, IP, and listening port.* Consumers, on the other hand, send discovery requests to subscribe to services of interest. When a provider is detected, the Routing Manager establishes a connection between the two peers, enabling method calls, property reading/writing, or the reception of asynchronous events.

The entire middleware configuration — from the services offered and consumed, to the discovery policies and network parameters — is defined in the *JSON* configuration files.

6.3.1 Provider implementation

For each control unit modeled in PREEvision as a **provider**, there is a *C++* application suited to offering the expected services. Implementation is based on the correct initialization of services via the vSomeIP API. The main features implemented are described below.

1. **Offer of a service**: the first operation performed by the program is to register the service using the function:

   ```
   app_->offer_service(service_id, instance_id);
   ```

   This call notifies the Routing Manager (local or remote) that the application is ready to offer a specific service, identified by its *service ID* and *instance ID*. From this moment, consumers can discover and connect to the service.

2. **Offer events**: after offering the service, the application offers the events associated with the service using:

   ```
   app_->offer_event(service_id, instance_id, event_id,
      its_groups, ...);
   ```

   For each event (identified by *event_id*) defined in the *JSON* file, this call allows the provider to enable the sending of asynchronous notifications, also declaring membership of an Event Group (*its_group*).

3. **Sending notifications**: payloads are sent periodically to consumers in the `notify()` function, in a dedicated thread. For each event, a payload is dynamically constructed according to its type (integer, string, boolean, etc.) and sent with:

   ```
   app_->notify(service_id, instance_id, event_id, payload);
   ```

   In the case, for example, of the `newBehaviorMonitoringDetected` event, the program simulates realistic scenarios of irregular driving or abnormal conditions, generating dynamic *JSON* messages and broadcasting them to registered consumers.

4. **Integration and cyclic behavior**: The program follows a cyclic logic, alternating between *“offer”* and *“stop offer”* states to simulate dynamic provider behavior.

Figure 6.5 shows the sequence diagram for the behavior of a **Provider** application. In addition to the main calls described, the diagram also shows how the subscription phase is handled by the **Consumers**.

In particular, the Provider receives **SUBSCRIBE** messages and upon receipt of a subscription request, the Provider responds with a **SUBSCRIBE_ACK** message, confirming that the Consumer’s interest in one or more events has been registered.

Only after this interaction asynchronous communication can start.

6.3.2 Consumer implementation

To simulate a consumer control unit, however, a *C++* application capable of using the required services was developed. The main features implemented are described below.

1. **Event request**: the following function allows the consumer to express interest in a specific event offered by a service (after first checking that the service is available). The method accepts the *service ID*, *instance ID*, *event ID*, and associated *event group* as parameters.

```
app_->request_event(service_id, instance_id, event_id,
    event_group, ET_FIELD);
```

   In addition, the type of event is specified. In this case, it is the `ET_FIELD` option, which indicates that the event will only be notified if the content (payload) changes, reducing the frequency of unnecessary messages.

2. **Subscribing to an event**: next, the consumer can subscribe to the desired event group, by specifying the *event group ID*, using the function:

```
app_->subscribe(service_id, instance_id, event_group_id);
```

   This two-step process — request event followed by subscribe — is necessary for vSomeIP to activate the communication flow and allow the consumer to receive asynchronous notifications from the provider.

3. **Notification reception**: finally, when an event is actually notified, the middleware calls the `on_message()` function. Within this callback, the program receives the event via a message object, from which it extracts the binary payload and interprets it according to the expected type (integer, boolean, string, JSON, etc.).

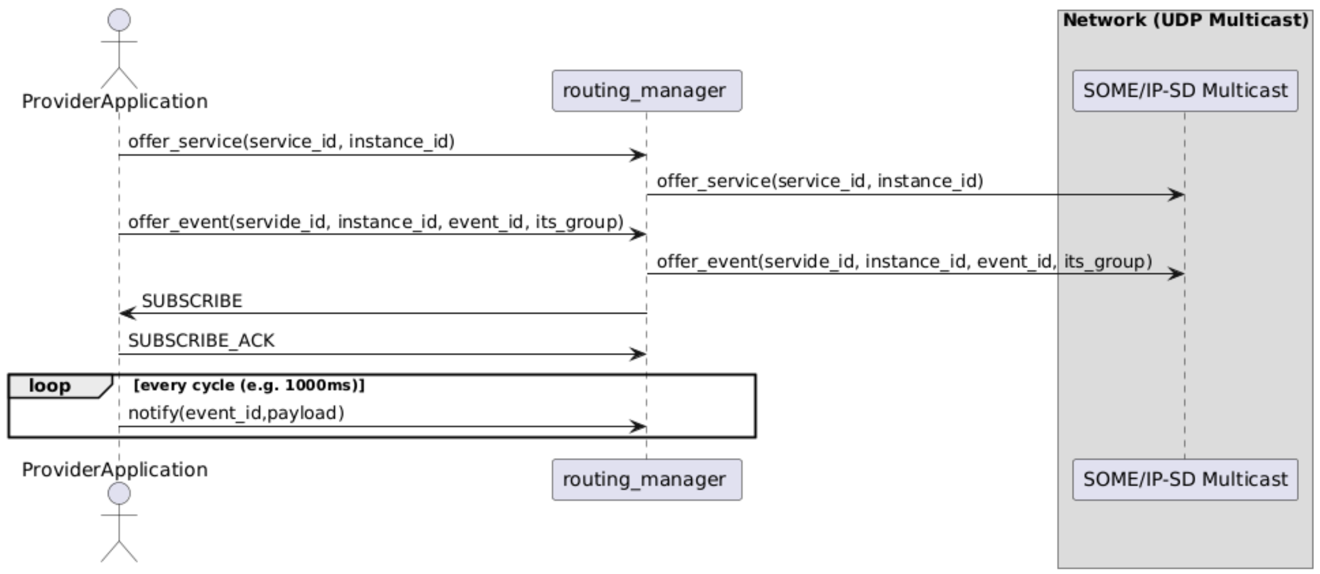

Network (UDP Multicast)
ProviderApplication
routing_manager
SOME/IP-SD Multicast
offer_service(service_id, instance_id)
offer_service(service_id, instance_id)
offer_event(servide_id, instance_id, event_id, its_group)
offer_event(servide_id, instance_id, event_id, its_group)
SUBSCRIBE
SUBSCRIBE_ACK
loop
[every cycle (e.g. 1000ms)]
notify(event_id,payload)
ProviderApplication
routing_manager
SOME/IP-SD Multicast

Figure 6.5: Provider sequence diagram

For example: if the name of the event is `newBehaviorMonitoringDetected`, a JSON object describing behavioral anomalies is extracted; for other events, it behaves accordingly.

Figure 6.6 shows the sequence diagram for the behavior of a **Consumer** application.

As shown in the diagram, following the `subscribe()` call, the Consumer receives a **SUBSCRIBE_ACK** message, which represents confirmation of the subscription by the Provider (consistent with the diagram in Figure 6.5).

Once this phase is complete, the Consumer application is ready to receive asynchronous notifications of subscribed events, which will then be processed within the `on_message()` handler.

This architecture allows to simulate in a realistic way the behavior of two control units capable of sending and receiving periodic or conditional events, as defined in the initial modeling on PREEvision.

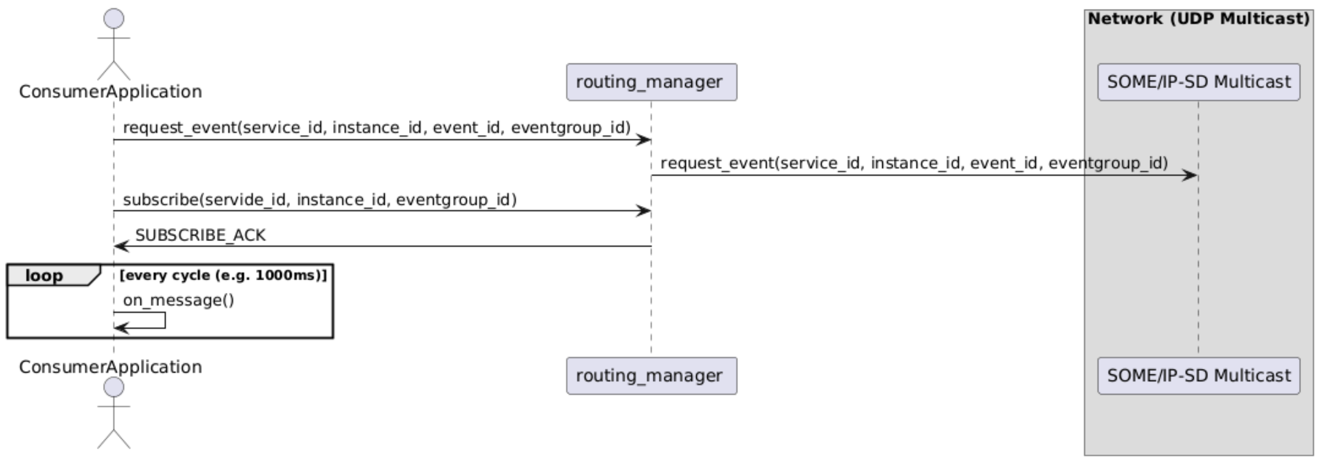

Network (UDP Multicast)
ConsumerApplication
routing_manager
SOME/IP-SD Multicast
request_event(service_id, instance_id, event_id, eventgroup_id)
request_event(service_id, instance_id, event_id, eventgroup_id)
subscribe(servide_id, instance_id, eventgroup_id)
SUBSCRIBE_ACK
loop
[every cycle (e.g. 1000ms)]
on_message()
ConsumerApplication
routing_manager
SOME/IP-SD Multicast

Figure 6.6: Consumer sequence diagram

Chapter 7

Conclusions

The thesis aimed **to explore and validate a model-based approach** suited for the design and prototyping of SOA architectures in the automotive sector. Starting from a model created in **PREEvision**, digital artifacts (*ARXML files*) were generated, which formed the basis for the concrete implementation of a communication system based on **vSomeIP middleware**. The transition from theory to practice was at the core of the work, demonstrating how it is possible to ensure **consistency** between the architectural definition and the runtime implementation of SOME/IP services on an Ethernet network.

The approach adopted fits into a context in which the complexity of modern vehicles requires tools and methodologies capable of **reducing errors, promoting collaboration, and improving the quality of distributed software**. Model-based systems engineering has confirmed these advantages, although it requires more time initially than more direct approaches. However, the time spent in the modeling phase leads to significant savings in the next development phases, reducing the likelihood of systemic errors and simplifying the analysis of overall behavior.

Among the main advantages found:

- Communication within the team is facilitated by the graphical and modular representation of the system;
- The localization of any errors was simple and immediate, thanks to the clear definition of interfaces and components;
- The models created were found to be reusable in different contexts, reducing the time needed for future designs.

7.1 Main outcomes

The thesis successfully demonstrated the possibility of **completing an entire design cycle**, starting from high-level modeling and ending with the concrete implementation of a functioning distributed communication system. The interfaces and services modeled in **PREEvision** were exported in *ARXML* format, from which a *Python parser* extracted and restructured the information in *JSON* format. These files were then used to automatically generate the *vSomeIP middleware* configurations, avoiding the introduction of manual errors.

Starting from these configurations, two distinct *C++ applications* were implemented — a provider and a consumer — capable of communicating with each other using asynchronous communication and event management typical of the SOME/IP standard.

The correct functioning of the infrastructure was **verified both in a simulated environment and on real embedded devices**, confirming the portability and robustness of the proposed approach.

One of the most important results of this work is the perfect **consistency maintained between the modeling in PREEvision and the implementation in code**. All *service identifiers*, *events*, *methods*, *event groups*, and *properties* defined during modeling were correctly processed and used in the implementation. Alignment with the specifications was ensured thanks to the intermediate *JSON* structure, which acted as a bridge between the architectural description and the configuration files required by *vSomeIP*.

The system was able to correctly start service discovery, subscribe to events, and receive notifications consistent with the expected behavior. Asynchronous event management demonstrated the reliability and efficacy of the middleware and the validity of the SOA approach even in realistic scenarios.

The use of *vSomeIP* was a strategic technical choice: the middleware offered features such as **Service Discovery, Routing Manager, and event management**, faithfully replicating the dynamics expected in a real distributed architecture and demonstrating high compatibility with the modeling obtained in PREEvision.

The clear separation between service definition and application logic made it possible to create a **modular**, **flexible**, and easily **extensible** system. The entire workflow was found to be automatable and adaptable even to wider industrial contexts, with a potential positive impact on the entire development cycle.

Another significant result of this thesis is the optimization of the architectural development and validation process. Thanks to the direct approach it was possible to **bypass many of the typical stages of the traditional automotive chain**. In particular, costly steps such as interaction with OEMs, involvement of Tier-1 suppliers, or access to industrial toolchains that can slow down the development cycle have been avoided.

This has made it possible to perform **internal benchmarks** in extremely short times, enabling rapid iteration on architectures and their conceptual validation without having to wait for external partners to take action. In quantitative terms, it is estimated that this approach can reduce the time needed to obtain a **working proof-of-concept** considerably, halving it in some cases. This leads to concrete advantages in terms of time-to-market, rapid prototyping, and agile innovation.

This simplification of the process allows:

- to test new architectures in-house independently;
- to validate the correctness of communication between ECUs already in the design phase;
- to validate architectural concepts before involving external suppliers.

In summary, the proposed approach allows to move more quickly **from idea to prototype** while maintaining methodological rigor and consistency. This represents an important step forward in the development of software-defined vehicles, where flexibility, modularity, and continuous verification are essential requirements.

7.2 Limitations and improvements

Although the work has achieved concrete and satisfactory results, some limitations have emerged that could provide ideas for future research.

Firstly, the absence of a formalized test environment is a critical issue. Although the system has been validated on real embedded devices, no objective metrics on latency, reliability, or throughput under high load conditions have been acquired. The definition of a test scenario would allow for a quantitative assessment of the infrastructure's behavior in realistic contexts.

7.3 Future work

The work opens numerous prospects for further development. An initial idea could be to extend the current implementation, which has so far focused on asynchronous events, to also include client/server mechanisms based on GET and SET methods. This extension would allow for even more complete simulation of bidirectional communication between ECUs, bringing the prototype even closer to a real-world scenario.

Other developments concern integration with **more complex simulation environments capable of reproducing network traffic and computational load conditions**. Furthermore, the adoption of automatic validation techniques — for example, through metrics performed directly on PREEvision models — would improve consistency between specification and implementation and further reduce the margin of error.

In a wider perspective, integration with emerging technologies such as the adoption of **alternative middleware** such as DDS or MQTT in collaborative and distributed contexts could expand the framework's field of application and make it suitable for areas beyond traditional automotive.

A first concrete step in this direction has already been taken with the modeling, within PREEvision, of the *Get Zonal Info* service, discussed in the Section 5.5.1.

The main goal is to enable the system **to evaluate the computational load and malfunctions of each ECU in real time**. This makes it possible to determine whether specific software services can be offloaded or migrated in response to ECU failures or overload. This functionality opens the way for more flexible and resilient architectures, in which service transfer is determined by the runtime conditions of the system.

From a wider perspective, such a mechanism also lays the foundation for a **cloud-based architecture** that can host additional automotive services externally or where services could be partially or temporarily offloaded to external infrastructures. In this vision, the vehicle becomes part of a broader distributed ecosystem in which services can be provisioned on demand, allowing the vehicle's capabilities to evolve and expand over time without requiring hardware modifications. By offloading selected functions to the cloud when necessary, it is possible to enhance performance, optimize resource usage, and introduce new features post-sale, enabling a truly software-defined vehicle paradigm.

In such a scenario, the exchange of information on the status and availability of resources between control units becomes not only useful but essential to ensure efficient orchestration and optimal performance in distributed automotive systems.

Each service has been associated with **custom attributes**, discussed always in Section 5.5.1, which represent the fundamental metadata for enabling runtime decisions. This can be exploited in the future for the implementation of a dynamic service relocation system.

This system, inspired by load balancing and fault tolerance principles, aims to improve the resilience of the zonal architecture by allowing automatic reassignment of services in case of overload or malfunctions. This provides a solid base for the future development of autonomous service orchestration mechanisms, which will become more central to the evolution towards SDVs.

Bibliography

[1] S. Arole, *“From Monolith to Service-Oriented Architecture: A Model-Based Design Approach towards Software-Defined Vehicles”*, Wipro Limited, Bangalore, India, 2022.

[2] A. Vetter, P. Obergfell, H. Guissouma, D. Grimm, E.Sax, M. Rumez *“Development Processes in Automotive Service-oriented Architectures”*, MECO, Budva, Montenegro, 2020.

[3] P. Obergfell, S. Kugele, E. Sax, *“Model-Based Resource Analysis and Synthesis of Service-Oriented Automotive Software Architectures”*, ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), 2019.

[4] GuardKnox, *“Automotive SOA for Software-Defined Vehicles”*, 2025. URL: `https://www.guardknox.com/automotive-soa/`.

[5] GuardKnox, *“Distribuited automotive SOA service for automotive infrastructure”*, 2020. URL: `https://blog.guardknox.com/distributed-soa-service-for-automotive-infrastructure`.

[6] IBM Documentation, *“SOA - Service Oriented Architecture”*, 2025. URL: `https://www.ibm.com/docs/it/baw/22.0.x?topic=designer-service-oriented-architecture`.

[7] MeshIQ, *“A Service-Oriented Architecture for the Automotive Industry”*, 2022. URL: `https://www.meshiq.com/a-service-oriented-architecture-for-the-automotive-industry/`.

[8] Amazon Web Service, *“The Difference Between Monolithic and Microservices Architecture”*, 2025. URL: `https://aws.amazon.com/it/compare/the-difference-between-monolithic-and-microservices-architecture/`

[9] Elettronica News, *“La tecnologia Ethernet nelle architetture a zone”*, April 2022. URL: `https://www.elettronicanews.it/tecnologia-ethernet-nelle-architetture-a-zone/`

[10] InsideEVs Italia, *“Auto elettriche: architetture zonali per ridurre peso e costi”*, May 2024. URL: `https://insideevs.it/news/725936/auto-elettriche-architetture-zonali/`.

[11] T. Smishad, *“Applying the V-Model in Automotive Software Development”*, eInfochips, 2021.

[12] Aptive, *“What Is the V-Model in Software Development?”*, March 2023. URL: `https://www.aptiv.com/en/insights/article/what-is-the-v-model-in-software-development`.

[13] N. Shevchenko, *"An Introduction to Model-Based Systems Engineering (MBSE)"*, Carnegie Mellon University, SEI, 2020. URL: `https://insights.sei.cmu.edu/blog/introduction-model-based-systems-engineering-mbse/`.

[14] J. Schäuffele *"Design and Optimization of E/E Architectures: PREEvision as Model-Based Tool for the Next Generation"*, 2016

[15] Vector Informatik, *"PREEvision Documentation and Product Information"* URL: `https://www.vector.com/preevision`

[16] AUTOSAR, *"AUTomotive Open System ARchitecture"*, URL: `https://www.autosar.org/`

[17] COVESA, *"The SOME/IP protocol"*, White paper, March 2025.

[18] Embitel Technologies, *"How SOME/IP Enables Service Oriented Architecture in the New Age ECU Network"*, 2020 URL: `https://www.embitel.com/blog/embedded-blog/how-some-ip-enables-service-oriented-architecture-in-ecu-network`

[19] MicroSys Electronics GmbH, *"miriac® SBC-S32G274A"* URL: `https://microsys.de/products/miriac-sbc-s32g274a/`

[20] Allion, *"Are You Ready For Automotive High-performance Computing in Future Smart Cars?"*, URL: `https://www.allion.com/tech_auto_smartcar_hpc/`.

[21] Vector, *"AUTOSAR Adaptive in PREEvision"*,Webinar Vector, March 2019.

[22] C. Harris, Atlassian, *"Microservices vs. monolithic architecture"*, URL: `https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith`.

[23] AUTOSAR, *"SOME/IP Protocol Specification"*, Release R22-11, November 2022, URL: `https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPProtocol.pdf`.

[24] AUTOSAR, *"SOME/IP Service Discovery Protocol Specification"*, Release R22-11, November 2022, URL: `https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf`.