Production-Optimized Software Architecture for Automotive Industry Based on AUTOSAR Classic

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

angenommene

DISSERTATION

von

M.Sc. Yi Zhai
geb. in Shanxi, China

Tag der mündlichen Prüfung:	08.05.2026
Hauptreferent:	Prof. Dr.-Ing. Eric Sax
Korreferent:	Prof. Dr.-Ing. Tobias Düser

Abstract

In recent years, the significance of in-car software has grown substantially in terms of the quantity and frequency of software updates. However, the rapid evolution of software often incurs challenges across its life cycle, particularly within automotive production. The stability and efficiency of automotive production are compromised due to the need to install the latest software packages and commission functions on assembly lines. Issues provoked by in-car software such as prolonged in-line flashing times and failed commissioning procedures can lead to manual workarounds and unplanned production stops. Consequently, car manufacturers are facing the ongoing challenge of balancing evolving software systems with production capabilities while adhering to cost targets.

This thesis elaborates on the challenges associated with microcontroller-based in-car software, which predominantly developed within the AUTOSAR Classic Platform (CP), from a manufacturing perspective. It highlights a gap in existing software architectures: the insufficient consideration of production requirements.

To bridge this gap, the thesis introduces Production-Oriented Software Partitioning (POSP), a novel software distribution based on the CP that physically isolates production-related software. Under the proposed distribution, the software is divided into a stable basic part for production usage, and one or more customer parts, which implement frequently evolving features. These parts can be compiled and installed independently, enabling flexible software deployment. A potential application is to flash only the basic part during production instead of the entire software system. This usage scenario can reduce the duration of in-line flashing by lowering the software size and ensure that commissioning and testing results remain independent from frequently changing customer parts.

The thesis further elaborates on the characteristics of POSP, including the need for inter-part communication interfaces, arbitration mechanisms to resolve conflicting outputs, and the additional resource. Additionally, a systematic methodology is proposed to refactor legacy software systems into POSP. A redundancy-based partitioning approach is proposed to handle signal-based, model-driven software systems.

The concept and methodology are validated through a prototype implemented on an industry-grade microcontroller running a fully functional CP platform. This demonstration proves the feasibility of POSP and the potential of series-production integration. Experimental results show a notable reduction in the software size required during in-line flashing and improved reliability during commissioning and testing within automotive production.

Kurzfassung

In den letzten Jahren ist die Bedeutung der Fahrzeugsoftware sowohl bezüglich der Menge als auch der Frequenz von Softwareupdates stark gestiegen. Allerdings bringt diese rasante Evolution der Fahrzeugsoftware häufig Herausforderungen im Software-Lebenszyklus mit sich, insbesondere während der Fahrzeugproduktion. Die Stabilität und Effizienz der Produktion können beeinträchtigt werden, weil die aktuelle Software auf den Steuergeräten installiert und immer mehr Fahrzeugfunktionalitäten während der Endmontage in Betrieb genommen werden müssen.

Dadurch entstehen Herausforderungen wie zum Beispiel verlängerte In-Line-Flashing-Zeiten und fehlgeschlagene Inbetriebnahmeprozesse, die zu manueller Nacharbeit und ungeplanten Produktionsunterbrechungen führen können. Dies stellt die Hersteller vor die Herausforderung, die sich immer schneller entwickelnden Softwaresysteme mit der Produktionskapazität und den Kostenzielen in Einklang zu bringen. Die vorliegende Arbeit erläutert die aktuelle Herausforderung bezüglich der mikrocontroller-basierten Fahrzeugsoftware, die überwiegend in der AUTOSAR-Classic-Plattform (CP) entwickelt wird, aus Sicht der Fahrzeugproduktion. Daraus zeigt die Arbeit die bestehende Lücke in Bezug auf eine produktionsoptimierte Fahrzeugsoftwarearchitektur auf.

Um diese Lücke zu schließen, wird das Konzept der Produktionsorientierten Softwarepartitionierung (POSP) vorgestellt, eine neuartige Softwareverteilung basiert auf CP, die produktionsbezogene Softwareteile physisch isoliert. Innerhalb der vorgestellten Verteilung soll die Software in einen stabilen Basisteil und mehrere Kundenteil aufgeteilt werden. Der Basisteil ist für den Produktionseinsatz vorgesehen und erfordert nur seltenere Aktualisierung, während die Kundenteile

sich häufig verändernde Funktionen implementieren. Diese Teile lassen sich individuell kompilieren und installieren und ermöglichen so eine flexible Softwarebereitstellung in der Endmontage. Beispielsweise kann nur der Basisteil anstatt des gesamte Softwaresystems während Endmontage aufgespielt werden. Dadurch kann die Dauer des In-Line-Flashings durch die reduzierte Datenmenge verkürzt werden und die Ergebnisse der Inbetriebnahme von den sich häufig verändernden Kundenteilen entkoppelt werden.

Darüber hinaus untersucht die Arbeit Eigenschaften von POSP, darunter den Bedarf an Inter-Teile-Kommunikationsschnittstellen, Arbitrationsmechanismen zur Auflösung von Konflikten zwischen überlagerten Ausgängen, und zusätzliche Ressourcen. Ferner wird eine systematische Methodologie vorgestellt, mit der sich vorhandene Legacy-Systeme mit Hilfe von POSP umwandeln lassen. Hierfür wird eine redundanzbasierte Software-Trennungsmethode vorgestellt, die signalbasierte, modelgetriebene Softwaresysteme berücksichtigt.

Das vorgestellte Konzept und die eingeführte Methodologie werden durch einen Prototyp validiert. Der Prototyp wurde auf einem Mikrocontroller mit einer voll funktionsfähigen CP implementiert. Außerdem demonstriert der aufgebaute Prototyp die Durchführbarkeit und das Potential von POSP, kurzfristig in die Serienentwicklung integriert zu werden. Die Ergebnisse der prototypischen Umsetzung zeigen eine deutliche Reduktion der Datenmenge, die beim In-Line-Flashing übertragen werden muss, wenn während der Endmontage nur der Basisteil installiert werden muss. Zudem wird eine verbesserte Zuverlässigkeit während der Inbetriebnahme erreicht.

Danksagung

Mit dem Abschluss dieser Arbeit geht eine intensive und prägende Zeit meines Lebens zu Ende. Der Weg von den ersten akademischen Schritten bis zur Vollendung dieser Dissertation war geprägt von zahlreichen Herausforderungen, wertvollen Erfahrungen, und persönlichem Wachstum. Die Erreichung dieses Ziels wäre ohne die Unterstützung zahlreicher Wegbegleiter nicht möglich gewesen.

Mein besonderer Dank gilt meinem Doktorvater, Herrn Prof. Dr.-Ing. Eric Sax, für die exzellente Betreuung und das entgegengebrachte Vertrauen. Seine Anmerkungen und die detaillierte Durchsicht der einzelnen Kapitel haben maßgeblich dazu beigetragen, die Qualität dieser Dissertation zu sichern und zu schärfen. Darüber hinaus habe ich durch seine Unterstützung eine tiefe Wertschätzung für wissenschaftliche Denkweise gewonnen, die meinen weiteren Weg prägen wird.

Ebenso möchte ich mich bei Herrn Prof. Dr.-Ing. Tobias Düser für die Übernahme des Korreferats und das damit verbundene Interesse an meiner Arbeit bedanken. Mein Dank gilt zudem den weiteren Mitgliedern der Prüfungskommission: Herrn Prof. Dr.-Ing. Ahmet Cagri Ulusoy, Herrn Prof. Dr.-Ing. Marwan Younis, und Herrn Prof. Dr.-Ing. Martin Doppelbauer.

Die vorliegende Arbeit entstand während meiner Tätigkeit als Doktorand bei Mercedes-Benz AG in Sindelfingen. In diesem Rahmen möchte ich mich herzlich bei Herrn Dr. Michael Hahn und Herrn Mario Caggiano für die kontinuierliche Unterstützung und den menschlichen Rückhalt bedanken. Zudem danke ich Herrn Ralf Kniefert, Herrn Frank Homberger, Herrn Timmo Schmitz, Herrn Konstantin Bunke, und Frau Dr. Rose Sturm für ihr Vertrauen sowie für die Bereitstellung der Ressourcen und Möglichkeiten, die ein hervorragendes Umfeld für diese Arbeit geschaffen haben.

Ein herzlicher Dank gilt Herrn Daniel De Monte. Er hat bereits kurz nach meiner Ankunft in Deutschland an mein Potenzial geglaubt und mich darin bestärkt, dass ich eine Promotion erfolgreich meistern kann. Sein Vertrauen war vor zehn Jahren der erste wichtige Grundstein für mein heutiges Ziel.

Darüber hinaus danke ich Kolleginnen und Kollegen am Institut für Technik der Informationsverarbeitung (ITIV) sowie am Forschungszentrum Informatik (FZI). Obwohl meine Forschung primär in der Industrie stattfand, habe ich die gemeinsamen Treffen und den fachübergreifenden Dialog als bereichernd empfunden. Die besondere Atmosphäre und der motivierende Geist unter den Doktorandinnen und Doktoranden haben mich auch in intensiven Phasen der Arbeit immer wieder angespornt.

Mein tiefster Dank gebührt meiner Familie und vor allem meiner Frau He Zhao, die mir mit unendlicher Geduld und Unterstützung stets den Rücken freigehalten hat.

Leinfelden-Echterdingen, im Mai 2026
Yi Zhai

Contents

1 Introduction

In the last 40 years, the automotive industry witnessed the exponential growth of in-car software from tiny bits to over 100 million lines of code [8, 38]. The software is used in vehicle-centric function domains, like powertrain control, chassis control and safety system, as well as passenger-centric domains, e.g. multimedia/telematics, body/comfort and human-machine interface [44]. The software dictating systems in modern cars brings innovative functions on the one hand. On the other hand, the software driven functions demand careful design, implementation, validation and verification in an automotive-specific manner [145].

Regarding the rapid evolution, the automotive software is becoming increasingly complex, which is demonstrated through multiple aspects, including the quantity of code, i.e., over 10 million conditional statements and about three million functions, the interactions within the vehicle (7000 external signals among more than 150 Electronic Control Unit (ECU) [165], and the large scale of functional requirements (over 100,000 [8]).

Moreover, automotive software systems come in various configurations to meet the unique needs of individual customers. For example, the Mercedes-Benz C-Class offers various model versions, including a three-door, a five-door, and a coupe, each of which can be customized with a range of optional features, such as a navigation system, Advanced Driver-Assistance Systems (ADAS), and more. With the added options of alternative engines and customer preferences, the number of possible vehicle configurations exceeds millions [130]. In light of this customer preference, OEMs are confronting with the task of producing desirable vehicles efficiently with high quality. Moreover, the task must be equally considered in the development and production processes with every new generation.

As a result, the increasing amount and complexity of automotive software leads to a revolution in E/E automotive architectures [28]. The traditional topology of ECU in which the E/E components realize mostly independently or loosely interconnected functions is now being replaced by an integrated architecture where the software functions are decoupled from hardware and centralized in some control devices. Although centralization provides flexibility and optimization opportunity of software/hardware architectures, the shift of the paradigm requires more stringent control to meet the safety and real-time requirements of the automotive industry [45].

Besides, the Electrical and/or Electronic (E/E) components have a relatively short life cycle of around four years [38], while a vehicle's expected lifespan is more than 15 years. This disparity is particularly pronounced in the software components, which may have an even shorter life cycle of just four weeks [53, 113]. The reason behind this is multifaceted and includes the malleability of software, the need to address software bugs, raise customer satisfaction, speed up time-to-market, and comply with evolving regulations.

While more than 95% of the software updates post the Start of Production (SOP) are not caused by the change of production processes[1], these software updates can introduce side effects to the assembly procedures, as in-car software must be installed to corresponding ECUs, and interfaces must be provided to finalize vehicular functionalities before delivery.

Against the background, the relationship between the in-car software and the manufacturing process is hardly mentioned in the industry and academia, which lead to the motivation of this work.

[1] This figure represents a representative industry benchmark based on expert assessment. While specific percentages may vary by OEM, the trend underscores the decoupling of software release cycles from physical assembly line changes.

1.1 Motivation

While the in-car software systems are getting connected and associated, mechanical engineers were dedicated for over 100 years to render various subsystems in cars independent [38]. Optimizing the cost and risk of car manufacturing in the perspective of Original Equipment Manufacturer (OEM) could previously be achieved by outsourcing engineering and production activities to suppliers. Besides, only OEMs are responsible for the interplay and integration of hardware and software on the vehicular level. However, with software becoming the primary catalyst for automotive innovation and inter-device collaboration serving as the foundation of advanced vehicle functions, this optimization is no longer guaranteed.

On the end assembly line (see Figure 1.1), where different components and sub-assemblies are mounted on the painted car body, the in-car software influences the efficiency of the manufacturing. First, the software of ECUs is installed on the end assembly line. This allows more flexibility regarding last-minute-updates and function personalization. Second, the ECUs are adjusted on the assembly line to ensure performance. The calibrations of parameters depend on the in-car software. Third, the installation and the integration of vehicular components are verified and validated on the assembly line, which requires the software-driven systems to carry out specific actions. For example, an external testing device leverages diagnostic interfaces to transmit a command to the corresponding ECU, which is supposed to control intended actuators [102]. The performance of the relevant functions is verified according to the act of the actuators.

The ultimate objectives of the end assembly are quality, efficiency and stability. While efficiency determines the delivery quantity of OEMs, the disruptions during the assembly processes can negatively impact the profitability of the entire automotive industry value chain.

Considering the production line operates on a 60-second interval [85], even a brief interruption of just a few seconds on the premium car model assembly line can have significant consequences that may include costly delays and production bottlenecks. To ensure the stability and the efficiency of assembly lines, the

feedback of production staff from various Mercedes-Benz factories all over the world is continuously summarized.

Until July 2022, over 200 items have been summarized. Excluding the repetitions, 23 representing topics are collected and sorted into the following categories: software, hardware, assembly process, and organization/strategy. In case a topic is related to two categories, each relevant category gains an index of 0.5. The results (see Figure 1.2) show that the end assembly can be supported effectively by dedicated software optimization.

Thereby, the software-related topics from the survey can be further abstracted to three concrete issues:

(I_1) It is desired to have stable software for commissioning and testing within the end assembly which is independent of software releases.

(I_2) The software for production should be modified so that employees are protected from improper behaviors of actuators during the ongoing installation.

(I_3) The flashing time should be reduced as much as possible.

As stated above, the automotive software is modified frequently to meet customer expectations, which can lead to incompatibility between the latest software version and the given commissioning process of the production line. In principle, the assembly processes should be aligned to the software update. However, this congruence can hardly be guaranteed in practice as the related software developers

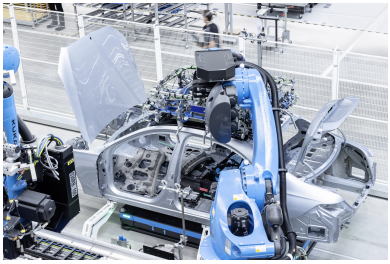


Figure 1.1: The Factory 56 of Mercedes-Benz in Sindelfingen, Germany. Source: Mercedes-Benz AG [104]

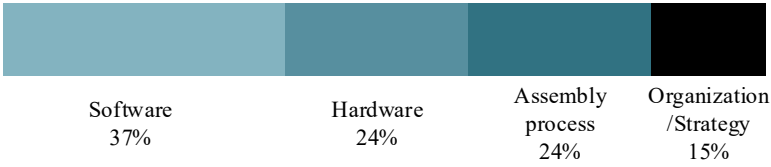


Figure 1.2: Categorization of collected requirements from assembly departments since 2021 (23 items in total)

are unfamiliar with assembly processes. Therefore, I_1 reveals the dilemma that the production is stability-oriented, but the software development and vehicular functionalities necessitate constant iteration.

For instance, a new software version for the 360° camera (see Figure 1.3) has caused mismatches with established commissioning and testing protocols. Due to customer feedback, a parameter in the camera software was changed. The performance improvement caused an unexpected error rate (about 10%) in the end assembly phase[2], as the images captured during testing in the pre-configured room differed from the desired results. The assembly staff was unaware that the software parameters were the root cause of the issue. To meet the testing standards, the "defective" cameras were exchanged manually.

Figure 1.3: The display shows the surrounding environment through the use of 360° camera, source: Mercedes-Benz AG [105].

[2] This figure is based on expert assessments and internal production reports in 2022.

I_2 refers to automotive actuators with the fail-safe feature but less intelligence, for instance, windscreen wipers and extractor fans for the engine. The devices are programmed to turn on their actuators, if they lose communication to upper machines. A windshield wiper running when not needed is less critical, than one not running when needed. On the contrary, the characteristic is unnecessary and even dangerous for the technicians working on the assembly line because the actuator can suddenly work during the commissioning process when the external testing device codes a connected ECU.

I_3 deals with the in-line flashing process, whose duration is becoming an increasing challenge. If the in-line flashing is not finished as planned, unfinished vehicles are caused at the end of the production line. These unfinished vehicles must be manually re-installed in parking lots, incurring additional expenses as special transportation is required to move the cars without driving capability from the end of the production line.

Besides, the prolonged boot-up time of the automotive infotainment system slows down the assembly process. The modern head unit in the car is powerful, but its boot-up time can take up to 60 seconds. Customers don't have to wait so long as the system can remain in sleep mode during inactivity or boot up in advance when the customer approaches the vehicle. However, the commissioning and testing processes in the end assembly necessitates the booted system and have to endure the booting period although most of the functions of infotainment are unnecessary for the assembly.

To tackle the software-related difficulties in the end assembly, various actions were taken, including the implementation of enhanced safeguarding processes for software release and the expansion of assembly lines to accommodate increased flashing time. However, there are two main drawbacks of these unilateral measures. Firstly, optimization of software release progresses increases the cost and does not provide a systematic approach to prevent accidents. Secondly, the renovation of the assembly line proves to be high-priced, often exceeding millions of Euros, and some assembly halls lack the necessary space for expansion or modification.

To summarize, the current trends in automotive software and the objectives of automotive manufacturing create a paradoxical scenario. While in-car software packages are growing to tens of gigabytes, the assembly line manager desires flashing time to be under one minute. While software upgrades and bug fixes are intended to be released monthly, a consistent commissioning and testing strategy during production period of the vehicle series is preferable for the end assembly hall. The conflicting objectives between in-car software and the end assembly process pose a significant risk to the automotive industry, and the problem cannot be adequately addressed from just the perspective of software or car manufacturing.

Therefore, the interest of automotive end assembly should be taken into account systematically. General strategies, approaches and methods on automotive software are required to ensure the interests of every participant along the life cycle of vehicles. The pursued quality goals, such as flexibility, modularity, and reusability, are valid for the software usage in the end assembly as well.

Software architecture is one of the cornerstones of successful products in the automotive industry [145]. To address the above-mentioned challenges properly, redesign and rethinking the automotive software systems in a high-level view is necessary. Hence, architectural approaches are expected to meet the software requirements which are complimented by the need for manufacturing. This doctor thesis concentrates on the interest of the end assembly, one of the most essential stakeholders and the first customer of automotive software and elaborates a production-optimized software architecture.

Especially, the doctor thesis concentrates on the Microcontroller (µC)-based automotive software systems, as the majority of vehicular control units are µC-based. Additionally, µC-based control units have direct interactions with sensors and actuators, who must be commissioned and tested within the end assembly. As most of the µC-based automotive control units are developed on the AUTOSAR Classic Platform (CP) [22] (detailed discussed in Section 3.3.1 and Section A.2), this thesis focuses on the software systems deployed on the CP.

This thesis is the result of research that is a part of the Software-Defined Car (SofDCar) project. The SofDCar project is carried out in a consortium from the

automotive industry and research institutes. The project is funded by the German Ministry of Business Affairs and Climate Control (BMWK). The project focuses on the challenges of vehicle E/E and software architectures and aims to provide new methods and processes to enable software updates and upgrades over the whole life span of vehicles [142].

1.2 Research questions

To design realizable software systems for µC-based automotive systems that address the interest of end assembly, it is necessary to establish the boundary conditions and requirements for the software architecture. This leads to the following research question:

> RQ_1: *What are the key characteristics of a production-optimized automotive software architecture that ensures efficiency and stability of the end assembly?*

Automotive software systems are subject to diverse requirements from multiple stakeholders and often have a long development history. As a result, many existing automotive control units operate with legacy software systems that have solid software architecture. Assuming that a derived production-optimized software architecture represents a desirable design pattern for CP-based automotive software systems, methods and approaches must be developed to refactor the legacy software system to align with the newly proposed architecture. Hereby, the second research question is formulated as follows:

> RQ_2: *How can the existing AUTOSAR Classic-based automotive software systems be refactored to align with the proposed software architecture?*

While a production-optimized automotive software architecture is derived from the interest and need of the end assembly, its validity must be evaluated in a production-relevant context. Additionally, implementing new architecture results

in changes of development methodologies and life cycle management of software systems. Considering the constraints and ecosystem of AUTOSAR Classic, these transitions lead to the formulation of the third research question:

> RQ_3: *What implications arise from the usage of the proposed software architectural changes within the AUTOSAR Classic software lifecycle, particularly for automotive production?*

The overall goal of this thesis is to establish a foundation for the automotive software development with practical insights. The intention is to imbue early in-car software design with a consideration for future utilization. The analysis and exploration of these posed research questions are to be conducted within the context of the automotive end assembly. The resulting framework can subsequently guide strategic decisions undertaken by automobile manufacturers.

1.3 Structure of the thesis

The structure of the doctor thesis is organized as follows. Chapter 2 lays the technical fundamentals relevant to this thesis. It begins with a brief introduction to the topic of automotive manufacturing, followed by an overview of the development process in the automotive industry. Additionally, the thesis delves into the automotive E/E systems, introducing the vehicular diagnostics within E/E systems.

Chapter 3 reviews the state of practice concerning the stability of automotive end assembly, emphasizing the influence of in-car software architecture. After a brief introduction to automotive software architecture, the chapter focuses on the AUTomotive Open System Architecture (AUTOSAR) platform that is widely adopted in the automotive industry. Furthermore, the chapter examines software refactoring techniques, leading to a discussion on potentials of a production-optimized automotive software architecture.

Chapter 4 introduces the concept of separating the interests of automotive production and software development by partitioning the software for the both

stakeholders, which corresponds to *RQ1*. Referred to as Production-Oriented Software Partitioning (POSP), the concept proposes to partition µC-based software systems to the basic and customer parts, where the basic part addresses the production usage, and the customer parts represents the interest of software development. Furthermore, the concept is explored with analysis of feasible distribution and boundary conditions of software design regarding embedded hardware.

To answer *RQ2*, Chapter 5 analyzes the legacy software systems of the µC-based automotive control units. Assuming that these legacy software systems are built by input, output, and application Software Components (SW-Cs) with diagnostic interfaces implemented within application SW-Cs, this chapter presents a systematic workflow to refactor the existing monolithic software systems into production-oriented partitioned systems. Additionally, the chapter highlights the strengths and limitations of the methodology.

Chapter 6 demonstrates the prototype implementation of the proposed concept. Two software systems are developed for vehicle power window functionalities within door modules using a Microcontroller (µC)-based system. The first system follows a conventional monolithic design process, while the second is derived from the monolithic system based on POSP. The prototype runs on an industry-grade µC platform equipped with a fully functional CP, demonstrating the feasibility of the proposed concept for series-production development.

Chapter 7 concludes the doctor thesis with a short summary and an outlook addressing the current limitations of the thesis.

2 Technical background

2.1 Automotive manufacturing

The automotive industry holds unique significance in the economy and in personal lives as means of transportation. Its impact on economic prosperity and job creation is evident across major markets and countries [120]. Within the European Union, approximately 12 million cars are manufactured annually, accounting for roughly 16% of the world's total production. Moreover, the automotive industry provides 2.6 million jobs, with an additional 10 million employment opportunities in related manufacturing throughout the European Union, including manufacturing of commercial vehicles and suppliers. The automotive industry's presence in Europe is further reinforced by about 300 production plants [2].

2.1.1 Automotive manufacturing systems and processes

In general, manufacturing encompasses the creation of artifacts, services, and objects to meet the needs of society. In a broader sense, manufacturing combines production factors to add value. For a first approximation and preliminary discussion, manufacturing can be seen as a process that utilizes energy, material, and information to transform inputs into objects [95].

Specifically, the automotive manufacturing activities in the view of Original Equipment Manufacturers (OEMs) can be analyzed on *systems* or *processes* as shown in Figure 2.1. Regarding manufacturing systems, the automotive manufacturing can be investigated from the three following perspectives [117].

The *structural aspect* considers the components of the production line, including machinery, material handling equipment, the labor resources, and the allocations to different activities. The *transformational aspect* covers the functional parts that convert raw materials into finished or semi-finished products. Typical activities of transformational aspects are stamping, casting and welding. The *procedural aspect* describes operating strategies and procedures, in which product planning and resource allocations are considered [117].

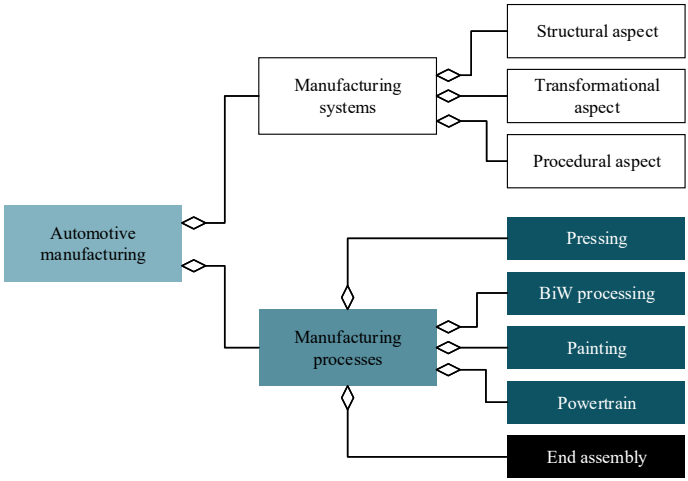


Figure 2.1: Overview of automotive manufacturing systems and processes, based on Omar [116].

In regard to *process*, the automotive manufacturing is elaborated in five sectors: *pressing plant, Body-in-White (BiW) processing, painting, powertrain*, and *end assembly* [72], as depicted in Figure 2.2.

In the pressing plant, the coil which is typically made out of steel and aluminum is transferred, washed, adjusted, pressed and stored (step 1 in Figure 2.2). Then, the sub-finished parts from the pressing plant are handled to body frames, namely BiW, which covers part detection, gluing, welding, and body cleaning (step 2). Thereupon, the finished BiW can be painted. The painting process encompasses pre-treatment, cathodic e-coating, sealing, and water treatment (step 3). At the

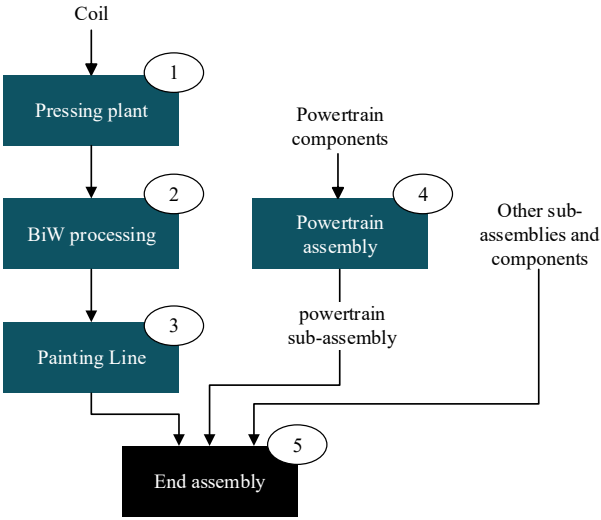


Figure 2.2: The processes in automotive manufacturing, based on Omar [116]

same time, the powertrain sub-assembly is facilitated (step 4), which handles the assembly of engines and gearboxes, heat treatment, leak tests, etc. Finally, all of the semi-finished parts of the vehicles are transported to the end assembly hall (step 5), where these parts are installed.

While the powertrain system and BiW mainly rely on in-house parts and components, the assembly plants are mainly dependent of the supply chain [117]. Nowadays, technologies and production of batteries and fuel cells are also seen as key areas of automotive manufacturing [120].

2.1.2 End assembly process

The end assembly is responsible for assembling more than 3000 interior and exterior components on the painted vehicle body [116]. Although varying for each car manufacturer, the end assembly process encompasses trim line, interior assembly, marriage, under floor assembly, and end-of-line test.

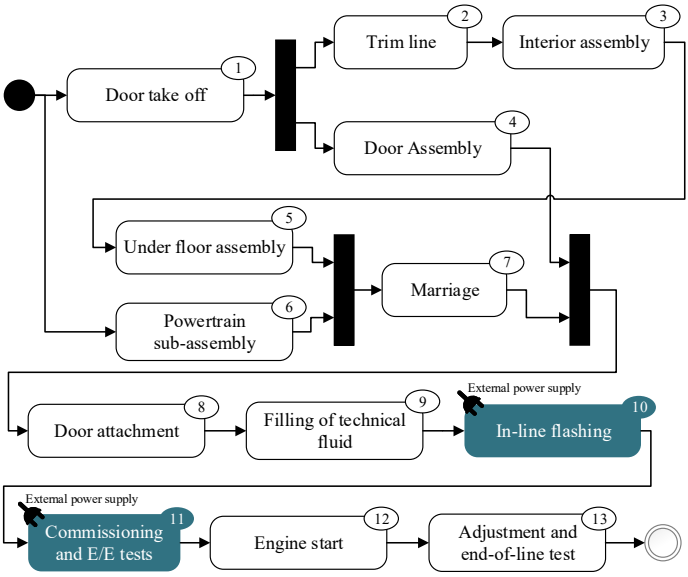


Figure 2.3: Important procedures in the automotive end assembly process, based on Omar [116]

Receiving the painted BiW from the painting line, the end assembly process begins with the removal of doors (step 1in Figure 2.3), which improves accessibility for subsequent assembly operations. After the doors are taken off, trims are affixed on the body (step 2), followed by the installation of interior components (step 3) such as the antenna module and the head unit. Then, door components like vehicular windows are mounted onto the door frames (step 4). This interior assembly precedes the underfloor assembly, where seats are installed (step 5).

Subsequently, the engine and transmission systems undergo assembly on a separate sub-assembly line (Step 6). Once assembled, these powertrain components are integrated into the vehicle body. This process is commonly referred to as *marriage* in the automotive industry (Step 7).

After marriage, the semi-assembled car body is no longer suspended, initiating the door attachment where the fully equipped doors are re-installed on the vehicle body (step 8). Reinstalling of doors is followed by filling of technical fluids (step

9). Specifically, refrigerant, windscreen wash water, and brake fluid are filled using the special operation modes of control units.

Hardware installation completion is followed by the in-line flashing (step 10), after which Electrical and/or Electronic (E/E) commissioning and testing procedures are conducted using the external testing device and power supply (step 11). The commissioning and testing procedures for the E/E systems are initiated and conducted manually by the assembly staff. This is necessary because certain vehicular actuators, such as LED strips and the sliding roof system, require manual verification for the functionality. However, the initiation of E/E commissioning and testing procedures will interrupt the in-line flashing process if it has not been completed.

Successful undercarriage E/E check enables the first engine start of the car's life cycle (step 12). The car is driven by an assembly operator from the production line. Subsequently, several calibrations and performance examinations are conducted, such as chassis setup, camera calibration, and leak test. Before delivery, the assembled car undergoes thorough functionality and appearance examinations via end-of-line tests, aided by the external testing device (step 13).

The role of software in automotive end assembly

The planning of end assembly highlights the significance of software-related factors in modern automotive manufacturing. During end assembly processes, the in-car software is directly involved in the following three procedures: *in-line flashing, commissioning,* and *testing*.

The usage of automotive software necessitates seamless installation of hardware components and sub-assemblies (step 1 - step 9 in Figure 2.3). Furthermore, the engine start (step 12) is scheduled immediately after the in-line flashing (step 10), commissioning, and tests (step 11). Any delay or failure in the flashing job or commissioning procedures renders the car undrivable at the end of the production line, leading to costly manual transportation and rework processes. The issue can

escalate quickly, given the production line's high running velocity of one car per minute [85].

Definition 2.1: Production relevance

Within in-car software, a software component (or SW-C) is considered production-relevant, if its functionality must be commissioned (step 11 in Figure 2.3) or tested (step 13 in Figure 2.3) during the end assembly.

Additionally, software-related processes play a crucial role in end-of-line control, specifically during the end-of-line tests (step 13), influencing procedures such as camera calibration, quiescent current measurement, and leak tests. These processes are intensively utilized to ensure the quality standards. The testing and commissioning results obtained from these software-driven procedures serve as evidence of the product's quality and performance.

Impact of software-related issues on end assembly costs

In assembly processes discussed above, the impact of software-related issues is reflected in the associated costs. These costs can vary across different production halls, as the planned length and duration for in-line flashing, commissioning, and testing processes differ from one production hall to another.

Typically, the in-line flashing process takes over 17 minutes and involves at least 30 control units. Additionally, the commissioning and testing procedures before engine starts can take another 15 minutes[1].

The financial estimation (see Appendix Section A.1) indicates the static costs of these procedures:

- Every minute of the in-line flashing costs approximately €0.26 for each car.

[1] The quantitative parameters (e.g., flashing time, number of ECUs) are derived from time analyses conducted within a high-volume automotive production environment in 2024.

- Every minute of commissioning or testing on the assembly line costs about €1 for each car.

The worst case that software may incur is unplanned production stop, or downtime, which can be caused by incomplete in-line flashing or repeated commissioning failures. Every minute of an unplanned production stop can result in losses up to €30,000, as discussed in Section A.1.

Additionally, it has been observed that approximately 1%[2] of cars produced on the assembly line require workarounds and analysis from the software development department due to unknown error codes encountered during production. Assuming the workaround of each car takes 30 minutes of an employee, which is often longer, the labor costs from these workarounds on a single production line can amount to €60,000 annually.

Furthermore, for about 10% of the produced vehicles, special measures such as ignoring certain error codes during assembly must be undertaken to avoid manual rework.

The financial impact is significant when considering the scale of production in the automotive industry. For example, reducing the in-line flashing duration by 10%, or 1.7 minutes, can lead to a corresponding reduction in the length of the production line. This efficiency gain would allow a single production line to save €106,080 annually. Such saving potentials highlights the meaning of optimizing automotive software to save every possible minute, or even second of the production process.

Therefore, the main objectives of the thesis are to reduce the in-line flashing duration while maintaining its stability, and to ensure predictable software behaviors throughout the end assembly to avoid unplanned production stops.

[2] This percentage is derived from the statistical analysis of production records over a multi-month period in 2023. It represents the ratio of vehicles that triggered a specific software issue escalation protocol to the total number of vehicles produced in the same period.

2.2 Automotive development processes

Error-free manufacturing in the automotive industry prerequisite correct and seamless development of vehicular elements. To ensure the result of developed vehicular systems, the development follows a well-planned development process (also product evolution process) that transforms strategic vision into the reality [171]. Within the development processes, the concept of system architectures plays a key role as the architecture defines the organization of system elements on each granularity level.

2.2.1 Automotive SPICE and V-model

In this thesis, the development processes of automotive electronic and software systems are considered in detail, as electronics and software are playing increasingly important roles in modern vehicles. The growing responsibility of electronic and software systems is made evident by the following factors [171]:

- Up to 40% of vehicles' costs are determined by electronics and software,
- 90% of all innovations are enabled by electronics and software, and
- 50 – 70% of the development costs for an ECU are related to software.

Against the background, the increased usage of software-related systems leads to an interest on optimization of electronic and software development processes and quality control of electronic and software systems. As a result, Automotive Software Process Improvement and Capability dEtermination (SPICE) is established and used by OEMs and suppliers to evaluate the development process of software-based systems in and around the vehicle [158]. Automotive SPICE is one variant of SPICE models. It is also utilized in aerospace and energy sectors.

Automotive SPICE assesses the capability of development processes using a two-dimensional assessment model, where the *capability level* is determined by

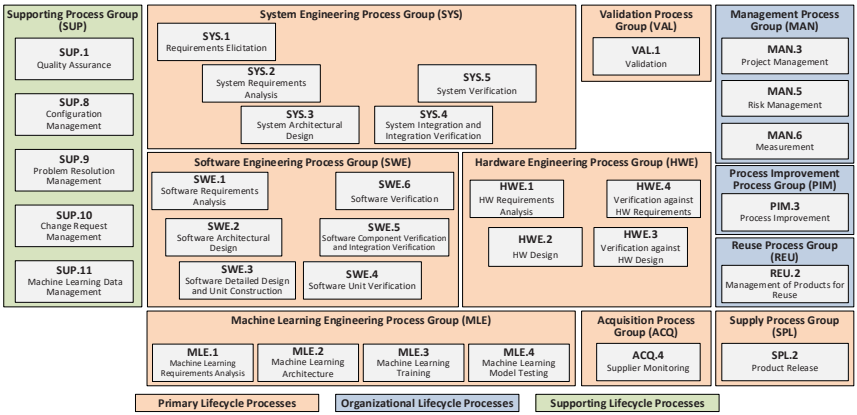


Figure 2.4: Overview of Automotive SPICE process reference model, based on VDA e.V. [158]

a measurement framework and the *process* is addressed by a process reference model [158]. An overview of the process reference model is provided in Figure 2.4.

Capability levels of automotive SPICE According to ISO/IEC 33020: 2019 [81], five *capability levels* are defined. As the capability level increases, the process becomes more structured and organized, enhancing the likelihood that development process outcomes can be controlled and predicted. The capability levels are accessed based on *base practices*, which indicate the extent of specific achievements, and *generic practices*, which measure the attainment of process attributes [158].

The capability levels can be explained as follows [81]:

- *Level 0* represents a non-implemented process or insufficient evidence of the achievement of the process outcomes. This level can be compared to uncooked food for cooking processes.
- *Level 1* process indicates specific defined outcomes, in which the base practices are implemented and work products are available. In comparison, kitchenware is defined, and the food quality is monitored and controlled.

- *Level 2* process is managed regarding activities and work products. Here, the food is delivered in time and according to a planned schedule.
- *Level 3* process is standardized, documented, and maintained, like cooking food according to a recipe.

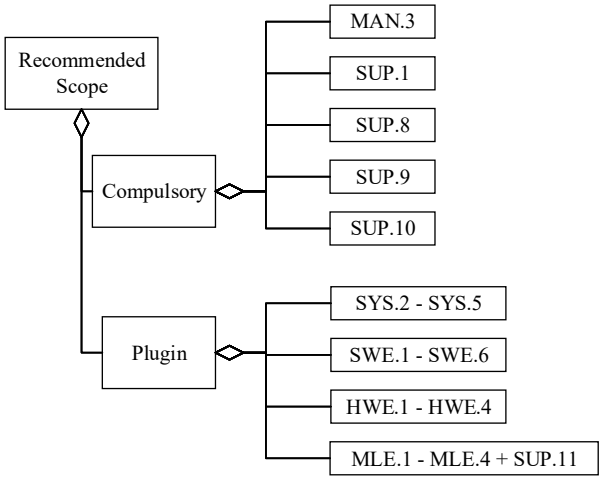


Figure 2.5: Recommended process scope of automotive system development, based on VDA [158].

Process of automotive SPICE The *process* reference model, shown in Figure 2.4, provides a set of processes, in which purpose statements are formulated and application environments are defined. To achieve corresponding capability levels, it is recommended to complete all the compulsory processes and at least one plugin, as illustrated in Figure 2.5.

For example, in the process MAN.3 (Project Management), base practices involve considerations and documentation on key topics such as:

- What is the content of the project handbook?
- How are measurable project goals defined?

- How are the status and progress of the project reported?

In addition to ensuring compliance with the mandatory processes (MAN.3, SUP.1, SUP.8, SUP.9, and SUP.10, as depicted in Figure 2.5), the development of automotive mechatronic ECUs typically involves both electronic and SW-Cs. This results in the usage of processes encompassing SYS, SWE, and HWE.

The SYS, SWE, and HWE processes in automotive software development align with the stages of concept, realization, integration, and validation. These processes are often represented using a V-model, as shown in Figure 2.6. In this model, the left-wing addresses requirements and implementation, while the right wing emphasizes verification and validation.

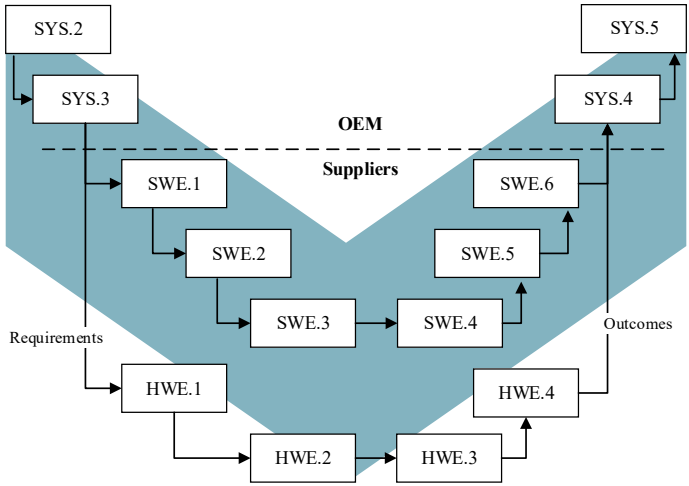


Figure 2.6: Processes of V-model within Automotive SPICE, based on Hindel et al. [68].

The OEMs are typically responsible for the system engineering processes, whereas suppliers concentrate on the detailed hardware and software realization, as illustrated by the dotted line in Figure 2.6. The progress starts with the requirement analysis of the system (SYS.2), which derives the system architecture (SYS.3). The designed system architecture is then substantiated by the OEM into hardware

and software requirements, which are further handed to suppliers. The OEM is not directly responsible for the software and hardware realization but for the integration of the system (SYS.4), once verified software and hardware components (SWE.6 and HWE.4) are delivered. Additionally, the system developed must be verified by the OEM (SYS.5).

Notably, the OEM does not take responsibility for software architecture design (SWE.2). Based on the illustrated process in Figure 2.6, the software architecture can only be influenced by the OEM indirectly through formulation of requirements.

Below the dotted line in Figure 2.6, the software and hardware development can occur in parallel and may even be undertaken by different vendors. Therefore, formulating precise and accurate requirements (SYS.3) and integrating software and hardware components (SYS.4) belong to challenges for automotive OEMs.

2.2.2 Vehicle system architecture

Realizing the recommended development processes according to Automotive SPICE necessitates methodologies of systems engineering. A basic idea of the development methodology within systems engineering is the usage of Requirement, Functional, Logical, and Physical (RFLP) perspectives.

The four perspectives of RFLP, shown in Figure 2.7, aim to ensure structured consistency during concept development. The F, L and P are the architectural perspectives which belong to the Vehicle Systems Architecture (VSA) [31, 90].

Figure 2.7 illustrates the concept of RFLP with usage of VSA, where the development of the automotive systems starts with the identification of stakeholder needs and use cases (step 1.1 in Figure 2.7). The interaction between the requirements (block 1) and functional VSA (block 2) facilitates the refinement of technical chains (step 2.2), stakeholder requirements (step 1.2), system use cases (step 1.3), and functional architecture (step 2.3). aid in identifying and clarifying development goals that are irrelevant to the technical solutions.

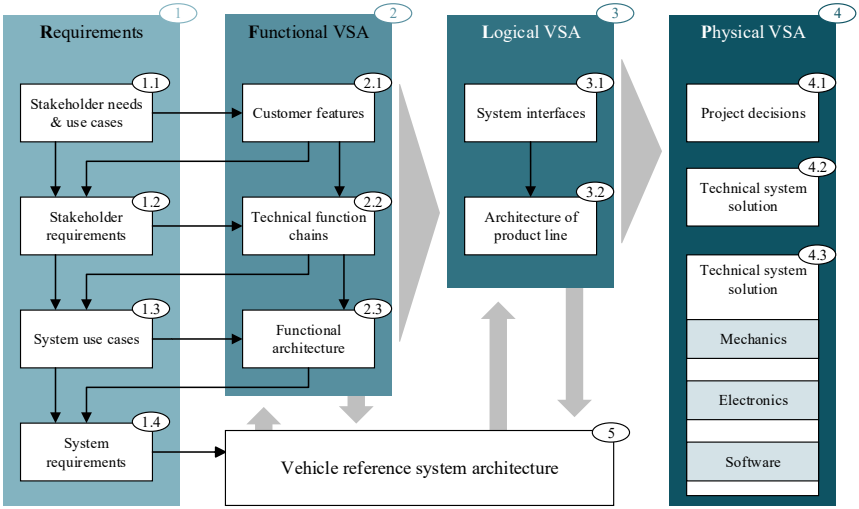


Figure 2.7: Conceptual Vehicle Systems Architecture (VSA) for vehicle concept development, based on Krog et al. [90]

Subsequently, the functional VSA leads to the logical VSA, wherein system interfaces (step 3.1) and product lines (step 3.2) are considered. The logical perspective involves the allocation of technical solutions to logical systems, representing a neutral architecture of technical solutions.

The physical perspective (block 4) addresses the current concept architecture development and the engineering disciplines with its technical solutions. This phase also encompasses the design and usage of E/E and software elements (step 4.3).

In parallel, the vehicle reference system architecture (block 5) represents a 150% model and serves as the base architecture for multiple vehicle derivatives. Consequently, it leverages synergies within the product portfolio, ensuring consistency and efficiency in development [90].

In this section, the terms introduced are summarized as follows. The process reference model of Automotive SPICE contains various process groups, notably the System Engineering Process (SYS) and Software Engineering Process (SWE). The

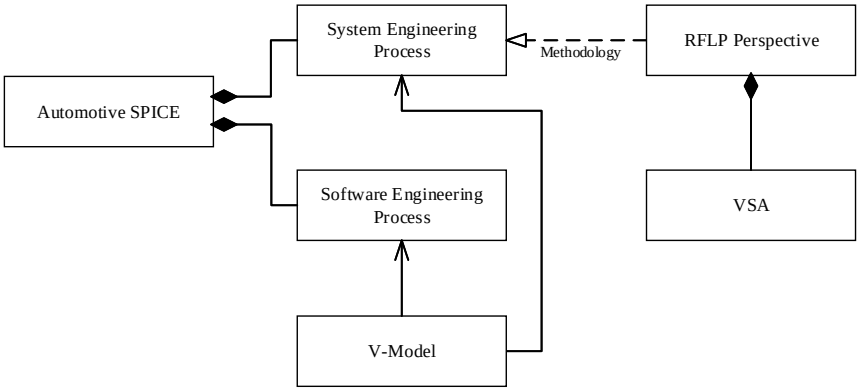


Figure 2.8: Relationship among terms around automotive development process

V-model is built upon the SYS and SWE. Additionally, Requirement, Functional, Logical, and Physical (RFLP) perspectives are utilized to illustrate the development methodology regarding systems engineering within the automotive industry. Finally, the Vehicle Systems Architecture (VSA) address the facets of F, L, and P within RFLP perspectives (see Figure 2.8).

2.3 Automotive Electric/Electronic systems

The ever-higher customer and legal requirements regarding the energy consumption, driving safety, and comfort lead to increasing usage of E/E systems in the vehicle. The demands on automotive E/E systems differ from other areas of consumer electronics and are characterized by the usage in changing environmental conditions, especially in terms of temperature, moisture, and vibration, high demands on reliability and availability, high safety and security requirements, and long product life cycles [132].

An automotive E/E system can encompass four types of elements: sensors, actuators, control units, and possibly setpoint generators. Information and energy are transmitted among these elements by the on-board wiring system [36].

Definition 2.2: Setpoint generator

Setpoint generators (*de: Sollwertgeber*) are parts of control or regulation loops. They are used to specify a desired setpoint either fixed or variable. Typical setpoint generators are buttons and knobs.

2.3.1 Features of automotive E/E systems

The automotive E/E systems are embedded, real-time and distributed systems.

Embedded systems are engineering artifacts designed to meet specific physical constraints while incorporating computational capabilities [67]. In the automotive industry, the E/E systems can be classified as embedded systems due to these physical limitations, such as limited installation space. Additionally, the embedded E/E systems within a vehicle are purposefully designed to perform dedicated tasks, and their functionalities are partially independent of user behaviors, e.g., the Antilock-Braking System (ABS) control [5].

In order to comply with safety regulations, critical vehicular functions driven by E/E systems should respond within a specified time frame. A real-time system is defined by its ability to guarantee a worst-case latency [157], with tasks categorized based on their activation time and deadline. The activation time is when a task is released, and the deadline is the latest time for task completion. Real-time requirements often define time windows for task execution, determined by the deadline and activation frequency [132].

Real-time systems can be categorized into hard and soft real-time systems. In a hard real-time system, exceeding time limits for response can have severe consequences, posing risks to safety and machinery [167]. Examples include the motor injection system and the steer-by-wire system [108]. Conversely, in a soft real-time system, violating execution times may result in a decrease in quality without causing damage or endangerment. Multimedia systems are an example of soft real-time systems [167].

To ensure that the real-time requirements are met and important tasks are executed within the given period, special mechanisms, such as interrupt and periodic pooling are employed in Microcontrollers (µCs). An interrupt refers to the automatic transfer of software execution triggered by a hardware event (e.g. rising or falling flanks of an electronic signal) that occurs asynchronously with the ongoing software execution, in which regular software functionalities are temporarily suspended and the interrupt service routine handles the specific requests associated with the interrupt event [157]. The priorities of different interrupt services are typically regulated by the interrupt vector table [96].

Modern cars incorporate a comprehensive network of over 150 control units [145], each designed to fulfill specific functions. These control units serve various purposes, ranging from specialized tasks to centralized computing and information transformation. The distributed E/E elements are connected to exchange information and to transfer energy. The increasing functional requirements on the car cause the growing number of E/E elements and bulky wiring harnesses. Therefore, to optimize the topology of E/E elements to reduce wiring harnesses and create new synergies from existing E/E elements, namely the evolution of automotive E/E architectures, is gaining interest from scientific and industrial fields.

2.3.2 Automotive E/E architectures

The automotive E/E architecture is a special viewpoint of the Vehicle Systems Architecture (VSA), see also Section 2.2.2, and can be defined as the fundamental organization of E/E elements in a vehicle to realize the desired function with emphasis on the interactions and interdependencies among the components, with the environment, and the principles guiding the design and evolution [83].

The VSA can be specified in the view of E/E-relevant elements, as shown in Figure 2.9. The development process follows RFLP perspectives, beginning with defining requirements, which are then decomposed into E/E functions. The E/E functions are correlated by the functional interfaces that are addressed in the logical aspect. The functions and interfaces are implemented by different SW-Cs, between

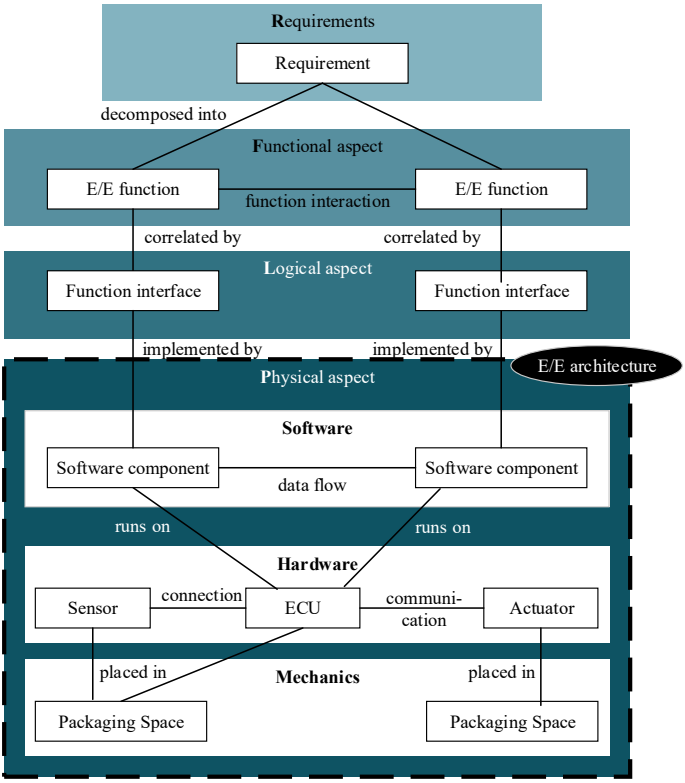


Figure 2.9: E/E viewpoint of the vehicle system architecture, based on the conceptual VSA shown in Figure 2.7 and Obergfell et al. [115]

which information is exchanged by data flows. Software components are distributed in different ECUs. To achieve the desired vehicular functionalities, measurement data is collected from sensors, while actuators are controlled by signals from the ECUs. Furthermore, the ECUs, sensors, and actuators are distributed in packaging spaces around the car.

As shown in Figure 2.9, the E/E architecture is seen as the allocation and interactions between ECUs, sensors, actuators, SW-Cs in the vehicle. Furthermore, we follow the decomposition method of Streichert et al. [147] that E/E architectures can

be investigated in levels of functional scopes, function allocation, networking topology, and hardware allocations.

Definition 2.3: ECU

In this doctor thesis, the term *ECU* is used to refer the circuit boards, the covering, the processing units, such as Microcontroller (µC) and microprocessor, and the auxiliary E/E elements, for instance, the power electronics and input/output interfaces.

Additionally, the term *hardware* in the thesis is used to refer the E/E elements, namely ECUs, sensors, and actuators in the vehicle.

Definition 2.4: HPC

Especially, *High Performance Computer (HPC)* platforms using microprocessors are utilized to implement functions in automobiles for future vehicle generation [134].

Specifically, automotive E/E architectures focus on the interactions and allocation of ECUs. Figure 2.10 illustrates the revolution/evolution progress of the E/E architecture, from a distributed approach to a integrated approach leading to fusion of ECUs and domains, which causes the usage of centralized vehicle computers [110].

The distributed architectural principle, as shown in style A in Figure 2.10, *one function - one box* is the result of function decomposition. The dedicated ECUs are connected to corresponding sensors and actuators. Limited data flow are executed by gateways between the ECUs in the distributed architecture. The increasing numbers of vehicular functions and requirements for quality goals [145], such as availability, safety, and reliability lead to the usage of system decomposition in the automotive industry. [132]. However, the one-to-one mapping causes large numbers of ECUs (approximately 150 [145]) and bulky wiring harnesses (more than 4 km [26]). Moreover, the traditional distributed approach suffers from restricted scalability and communication performance [28].

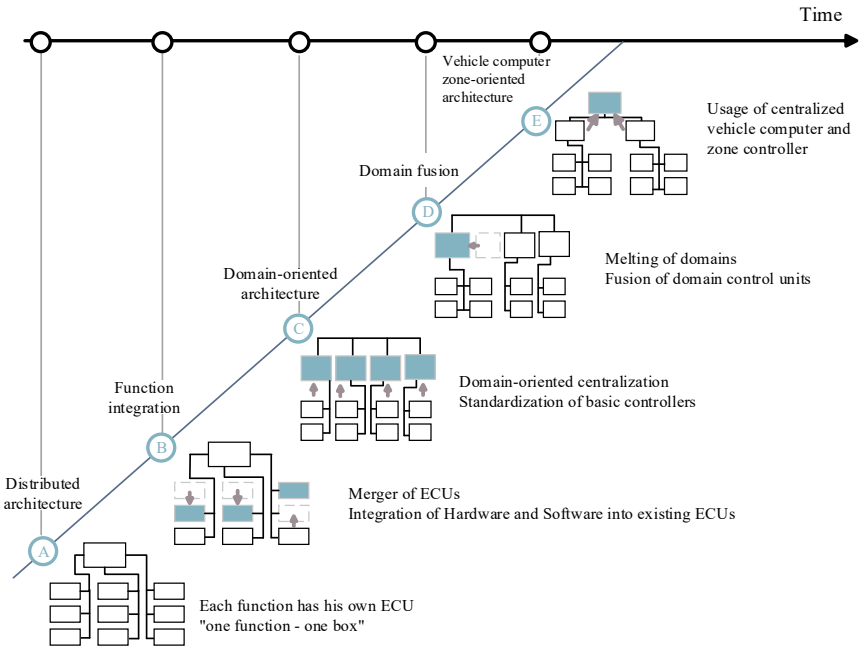


Figure 2.10: Evolution/revolution of E/E architectures, based on Di Navale et al. and Ren et al. [110, 126]

To address the drawbacks of the distributed architecture, the automotive industry has adapted E/E architecture towards centralized alternatives [28]. The centralization of E/E architecture in the automotive industry is inspired by the paradigm shift and introduction of integrated modular avionics in the avionics community [45].

ECU merging, presented as style B in Figure 2.10, was an early attempt to consolidate functions. However, domain-oriented architecture (style C in Figure 2.10) became prevalent in modern automotive E/E systems. This approach divides the system into four widely agreed-upon domains in the automotive industry: Body (comfort and lighting), Infotainment (displays, entertainment, and information systems), Motion and safety (chassis, active/passive safety, and driver assistance systems), and Powertrain (propulsion and exhaust gas treatment) [28]. As Figure 2.11 depicts, Mercedes-Benz adopted the STAR-3 domain-oriented

architecture, which utilizes the Ethernet backbone to decouple physical networks and enable inter-domain communication [133].

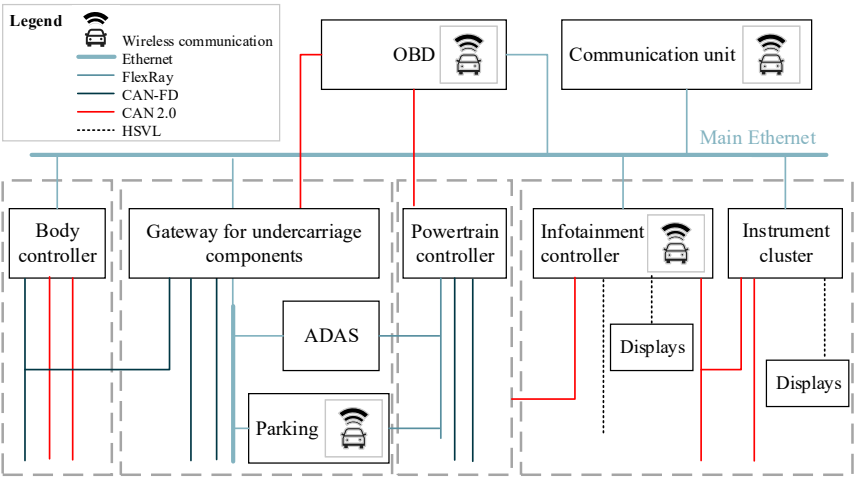


Figure 2.11: An overview of Mercedes-Benz STAR-3 E/E architecture, based on Scheer et al. [133].

However, despite its benefits, the domain-oriented architecture faces challenges in reliably implementing vehicular functions such as ADAS, which require close cooperation among diverse domains, such as ADAS and powertrain [62]. Furthermore, the inter-domain functionalities make the boundaries of domains unclear. These drawbacks of the domain-oriented architecture lead to the merging of domains, as shown in style D in Figure 2.10. The use of centralized vehicle managers and standardized input/output modules further enhances scalability, expandability, and openness. MAN, for instance, demonstrated the implementation of this concept, where ECUs no longer contain strategic functionalities [153].

At the macro level, the revolution of E/E architecture in the automotive industry is primarily driven by the demand for greater flexibility and inter-domain functionalities. However, at the micro level, the evolution of E/E architectures is influenced by cost factors and the distribution of labor. The reusability of control units, sensors, and actuators from predecessor models plays a significant role in shaping

these architectures. Additionally, integrating software and hardware from different vendors can slow down the centralization of E/E elements.

In the view of automotive manufacturing, the trends of centralizing E/E architecture brings both challenges and opportunities. On one hand, moving away from the *one function - one box* principle disables the ECU-oriented testing and commissioning approach, as production processes increasingly rely on the cooperation of multiple E/E elements. This necessitates linking phenomena with the underlying reasons, which cannot be limited to just one ECU. Moreover, the growing quantity of vehicular functions, with software encompassing over 100 million lines of code [44], leads to comprehensive testing and commissioning procedures during the end assembly. There are currently over 65000 different Diagnostic Trouble Codes (DTCs)[3] indicating the state and errors of the ongoing installation. Integrating and merging vehicular functions in centralized computing units introduces challenges in terms of error interpretation and localization.

On the other hand, centralizing E/E architectures accelerates the decoupling of software and hardware, as the centralized computing units no longer have direct access to controlled sensors and actuators. Clearly decoupling software and hardware allows for better tailoring of commissioning and testing procedures during final assembly to address hardware-related issues.

Given the background of the introduction of the Over the Air (OTA) approach in the automotive industry, OEMs are enabled to deploy software updates via wireless interfaces and internal vehicle bus systems[16]. The enhanced computing accelerates data transmission and reprogramming of ECUs. Consequently, the centralization of E/E architectures presents the opportunity to relocate the software installation process from the end assembly. This shift provides valuable flexibility in the in-line flashing procedure, as some parts of the software can be installed in the post-production phase.

[3] This figure reflects the technical specifications of a representative mass-production vehicle model released in 2023.

2.4 Vehicle Diagnostics

The role of vehicle diagnostics is identified by the task of identifying failure patterns and appropriate repair measures in workshops, eliminating the need for costly disassembly and maintenance [125, 175]. However, the utility of vehicle diagnostics extends far beyond the workshop setting and encompasses the entire life cycle of the vehicle.

During the vehicle's development, diagnostic functionalities are utilized to test components, subsystems, and the overall vehicle, evaluating their functions and properties as part of the function development process [32]. Additionally, off-board diagnostics are utilized in the in the verification and validation of E/E processes during end assembly.

Definition 2.5: Vehicle diagnostics

In this thesis, *vehicle diagnostics* are defined as a set of functionalities driven by in-car software which are designed to inquire about the states of vehicular functions, software, and hardware, and to execute dedicated actions accordingly.

Due to the usage of HPC platforms and centralized E/E architectures, there is a need to adjust and reshape vehicle diagnostics, as the current vehicle diagnostics are focused on hardware-related issues that are pre-defined by DTCs [32], which does not correspond to the incoming functionalities of the vehicle's life cycle. The decoupling of software and hardware in automotive E/E systems results in the necessity of software-oriented diagnostics.

2.4.1 On-board and off-board diagnostics

Regarding the function principle, vehicle diagnostics can be categorized into on-board and off-board approaches. On-Board Diagnostics (OBD) are the internal diagnostic system built in the vehicle, continuously providing the occupant and

technician with information about the vehicle's engine and operation [64], see Figure 2.13. More precisely, the OBD system monitors the emission relevant components or systems, stores detected malfunctions, and activates malfunction indicator lamp if necessary. Moreover, the detected states and errors are recorded in the non-volatile memory in the form of DTCs [50]. An example of DTC is shown in Figure 2.12.

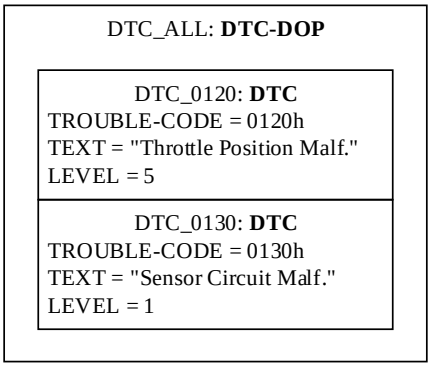


Figure 2.12: Example DTC Data Object Property (DOP), based on ISO 22901-1 [79].

In comparison, off-board diagnostics shown in Figure 2.13 involves connection between the on-board systems and external diagnostic tools (also referred to as external diagnostic device or external testing device in other literatures) to access and analyze the data, and sometimes change the parameters and running states of E/E elements.

Off-board diagnostics take care of the diagnostic functions of ECU functions other than emission, in which Unified Diagnostics Services (UDS) [82] is the most popular diagnostic protocol [50]. Thanks to the connected diagnostic tools, the capacity of off-board diagnostics is not restricted by the limited computing power and memories of ECUs in the vehicle. Therefore, off-board diagnostics are responsible for the following tasks [132]:

- Dedicated diagnostics for setpoint generators, sensors, and actuators,

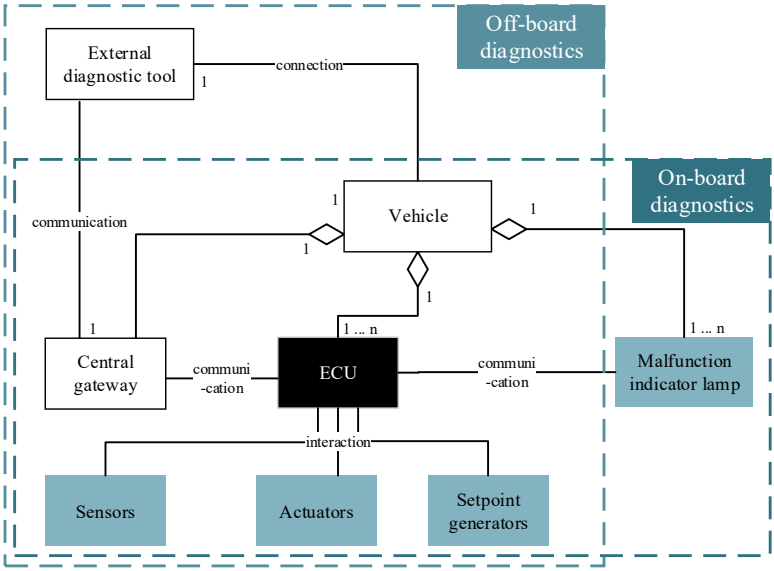


Figure 2.13: On-board and off-board diagnostics

- Runtime diagnostics using special algorithms, and
- Reading and clearing the non-volatile error memory.

In automotive end assembly processes, the software-related procedures are predominantly realized by off-board diagnostics. The external diagnostic tools are used to execute diagnostic tasks. For instance, the testers are used during the in-line flashing, commissioning, and testing (step 10, 11, and 13 in Figure 2.3).

2.4.2 Software update by flash programming

In µC-based embedded systems, there are two types of memory directly connected to or integrated in the processor: Random-Access Memory (RAM) and Read-Only Memory (ROM). While contents of RAM are erased if the system loses power,

meaning RAM is volatile, the parameters and running logic of the embedded system are saved in ROM which is non-volatile memory and stores data permanently [112].

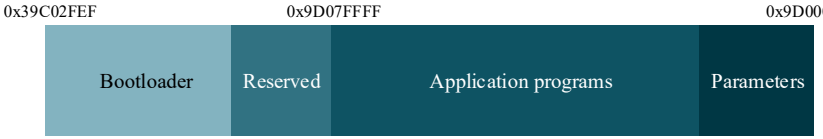


Figure 2.14: Flash memory organization of µC-based system, example from Microchip PIC32MZ devices [107].

Various types of ROMs are utilized in embedded systems, with Electrically Erasable Programmable ROM (EEPROM) being a widely used variant. A more cost-effective and faster alternative to EEPROM is *flash* memory, which allows data to be written and erased in blocks or sectors [112]. In automotive ECUs, the use of flash memory enables software updates through the off-board diagnostic interface. This method, known as flash reprogramming, allows for software updates without removing the control unit from the vehicle, resulting in significant cost savings compared to replacing the entire control unit [132].

Definition 2.6: In-line flashing

In this doctor thesis, *in-line flashing* is used to refer the flash reprogramming process of automotive ECUs during the end assembly.

The in-line flashing the flash memory of automotive ECUs that typically contains three types of editable data, as illustrated in Figure 2.14:

- *Bootloader:* The bootloader is an independent application from the main software programs and corresponding parameters. It is executed at the beginning of the system software and manages the applications without using a specified hardware interface, such as Joint Test Action Group (JTAG) [119].
- *Application programs:* Application programs represent the control logic for software applications other than the bootloader. In µC-based systems, flash

memory can be managed in different sections, accommodating multiple parts of application programs.

- *Parameters:* Software parameters are stored separately in blocks within the flash memory. These parameters encompass constants used in software applications, pointer addresses, mode and operating variables that should have initial values at the beginning of the application programs.

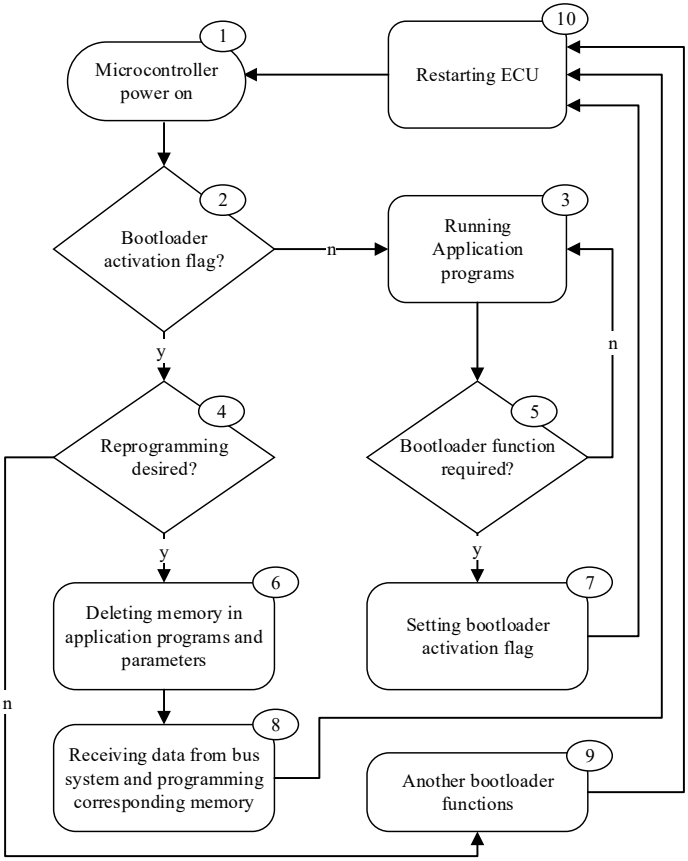


Figure 2.15: Generic flash reprogramming logic using a bootloader, based on the flowchart symbols [100].

Figure 2.15 illustrates a typical flash reprogramming process using a bootloader. The process begins with the supply power being given to the system (step 1 in Figure 2.15). Then, it is checked whether the bootloader should be started (step 2). If flash reprogramming is desired, the memory sections for application programs and parameters are deleted (step 6) and reprogrammed (step 8). The bootloader should include drivers for the corresponding protocol, such as UDS, as the function realization in application programs gets erased. After the data transmission and reprogramming are verified, the bootloader initiates a reset (step 10).

In normal operation of the embedded system (step 3), it is also possible to start flash reprogramming by sending a diagnostic request from the diagnostic tool. Once the bootloader activation flag is set (step 7), the µC is immediately restarted (step 10).

Several factors influence the efficiency of flash reprogramming. Firstly, the transmission velocity from the data source (e.g., the external diagnostic tool) and the volume of data packages affect the duration of flash reprogramming. Additionally, the capabilities of the µC processor, gateways, and overall E/E architecture can impact the flashing process. Since the µC is not directly connected to the external diagnostic tool (illustrated in Figure 2.11), any point along the communication path between the information source and the target device can become a bottleneck. The E/E architecture can influence the process, especially when the reprogramming processes of different ECUs are dependent on each other due to the network topology.

Figure 2.16 depicts a typical scenario of flash reprogramming in the automotive end assembly, which may result in manual rework. Reprogramming of domain-specific controllers must follow the flashing of domain controllers. Moreover, there may be flashing dependencies between domain-specific controllers. For instance, the entire in-line flashing process on the assembly line takes approximately 1000 seconds (discussed in Section 2.1.2), of which roughly 50 seconds are allocated as a buffer for emergency cases. In Figure 2.16, domain controllers are denoted by letter *D* while domain-specific controllers are labeled with letter *S*.

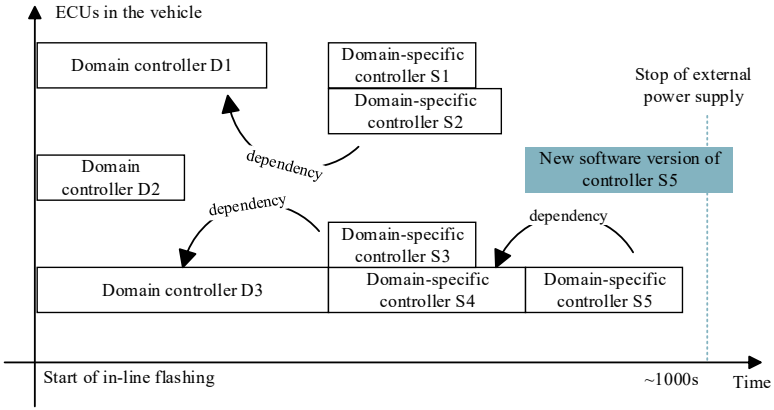


Figure 2.16: Example in-line flashing process for 3 domain controllers and 5 domain-specific controllers in an automotive E/E system using domain-oriented architecture.

Due to the capacity of gateways, the reprogramming of controllers S3 and S4 must wait for the deployment of controller D3, which slows down the entire process further. Furthermore, adding new vehicular functions can lead to enlarged software packages, as shown in the case of controller S5 in Figure 2.16, extending the overall duration of in-line flashing. If the in-line flashing job is not completed at the position stop of the external power supply, the new-built car cannot be driven from the production line, see also Section 2.1.2. The car has to be transported to the parking lot, and the corresponding flashing process has to be carried out again.

2.4.3 Diagnostic-based commissioning and testing procedures

The commissioning and testing procedures in the end assembly and in after-sales service are based on off-board diagnostics. While external testing devices for after-sales take the responsibility of flash reprogramming for ECUs and basic diagnostic procedures, such as deleting the fault memory, the testing devices in automotive end assembly have more comprehensive jobs. Besides carrying out the

hardware calibrations, the testing devices for the end assembly have user interfaces in the form of a display that supports the assembly staff to execute the tests properly.

Due to the limited memory in automotive ECUs, the *know-how* of commissioning and testing procedures locates in the external testing device. Correspondingly, the in-car software provides communication interfaces, with which the external testing device can read or write variables or memories. Typically, the commissioning and testing procedures in the end assembly follow the *action - polling* principle.

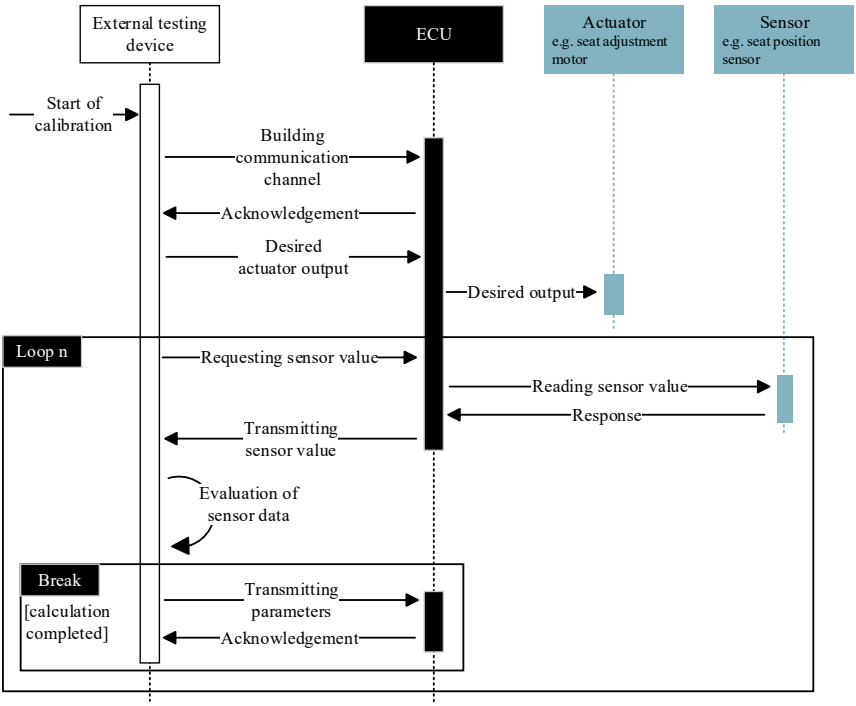


Figure 2.17: Sequence diagram for a typical calibration process for automotive ECU in the end assembly

Figure 2.17 depicts a general sequence outlining the progression of parameter calibration within automotive ECUs. The calibration process is initiated by an

external testing device, aimed at determining ECU parameters stored within the flash memory parameter section, as illustrated in Figure 2.14.

Once a secure communication channel is established, the targeted ECU transmits an acknowledgment signal back to the testing device. Subsequently, the testing device dispatches the desired actuator value for the calibration procedure. This action suspends the normal operations of the control unit, typically enacted within specific software modes, such as *production mode* and the extended diagnostic session. These specialized software modes facilitate direct manipulation of actuators and override the setpoint generator. The desired actuator output can be implemented by the method *Write Data by Identifier* of the UDS protocol [101].

Following this, the external testing device initiates a polling process to retrieve the value of the corresponding sensor. The ECU forwards the measured value to the testing device. This polling can be realized using *Read Data by Identifier* method of UDS protocol. The testing device evaluates the requested sensor value, with the polling process continuing until sufficient data is gathered. Once the required data is collected, the testing device deduces the corresponding parameters.

After this data aggregation, the testing device transmits these parameters back to the ECU. The ECU then proceeds to write these parameters into the flash memory by using the above-mentioned *Write Data by Identifier* service and confirms the successful transmission by emitting an acknowledgment signal.

Certain tests may prove challenging to execute solely through the connected sensors and actuators, particularly in cases involving factors like lighting systems and buttons. Consequently, specific testing and commissioning procedures during the end assembly necessitate the collaboration of assembly personnel guided by the testing device, as illustrated in Figure 2.18. While both actuators and assembly staff interact with the object, the testing device or assembly staff can subsequently evaluate these impacts.

In the view of testing algorithms, in-car software within control units must share a common *language* (the UDS protocol) with the testing device, as the data transmission, namely reading and writing, is based on mutually recognized

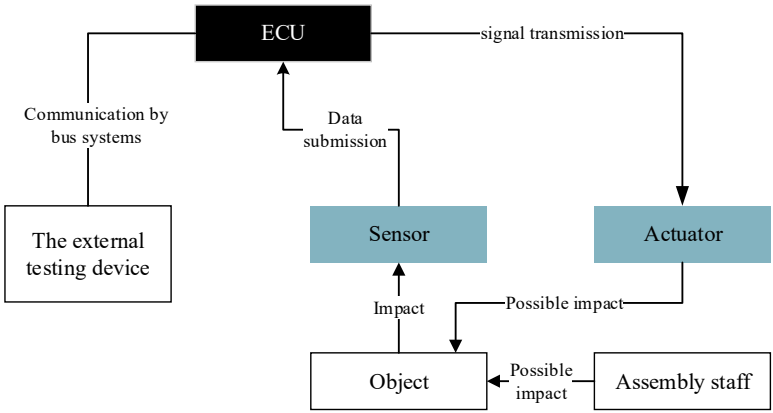


Figure 2.18: Possible participants in a testing or commissioning procedure of automotive end assembly

identifiers. Given the frequent software updates in automotive ECUs, two critical concerns arise with respect to this shared *language-based* communication:

- Discrepancies in *dialects* spoken by both parties (testing device and corresponding ECU). For instance, the testing device may request reading or writing data with incorrect identifiers, leading the control unit to abstain from executing the desired actions. This mismatch can result in improper commissioning or testing procedures.
- Even when both parties share the same *language*, misunderstandings can arise in interpreting the *words* (parameters or interfaces of the diagnostics) within the shared *dictionary* (OEM-specific diagnostic implementation). For example, a command like *Set motor velocity to default value* possesses maneuverability, and the default velocity may vary across in-car software versions. Consequently, undesired velocity can be measured during testing, potentially leading to an assessment of test failure, despite the motor's proper functionality and correct installation.

3 State of the Art and Potentials

3.1 State of practice regarding assembly stability

As previously discussed, various approaches are tried out to tackle the software-related difficulties in the end assembly, including the reconstruction of the assembly hall, refer to Chapter 1. However, these approaches have proven to be expensive and only partially effective in resolving the challenges associated with software usage within production.

As alternatives, strategies targeting software-related issues are introduced in this section. However, these strategies avoid direct intervention within the in-car software or its architecture. The reasons for circumventing are listed as follows:

- The complexity of in-car software is escalating, which is evidenced by the scale of large automotive Simulink models containing up to 15,000 building blocks, 700 subsystems, and 16 hierarchical levels [149].
- The software is developed based on rigorous processes without much consideration for production aspects, as outlined in Section 2.2. Software changes for production can lead to impairment of process safety or repeating development processes.
- The diagnostic interfaces (see also Section 2.4.3) are nested with other software functionalities. Software changes for production usage can incur undesired changes of other functionalities.

Specifically, the thesis examines and evaluates three state-of-practice methods that are currently employed to enhance the stability of end assembly.

3.1.1 Production-specific software versioning

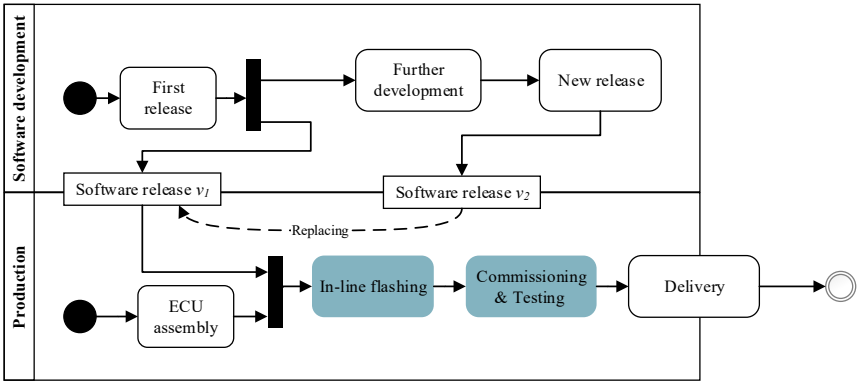
The software offered to customers must be up to date, but there is a distinction between customer-used software and the software used in end assembly. It is not compulsory for the software in end assembly to be the latest version. In fact, there has been practices utilizing a determined production-specific software version thorough testing and commissioning to ensure stability during production, while the software is further developed.

Concurrently, the software development department adheres to the development process outlined in Section 2.2, delivering released software (v_1) to the production. If new features need to be added, new functions are implemented, and the software is updated to a new release (v_2), replacing the previous version for in-line flashing within the end assembly as depicted in Figure 3.1-(a) and discussed in Section 2.1.2.

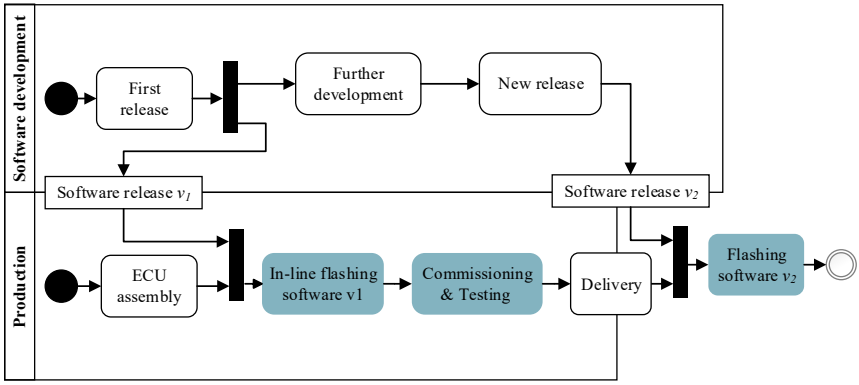
To prevent software-related issues during the end assembly, the thoroughly tested and released software version, referred to as the production-specific version, continues to be used. However, this usage leads to software re-flashing after the vehicle leaves end assembly, either at parking slots or in workshops, as illustrated in Figure 3.1. From the production perspective, this software appears to be frozen.

The approach production-oriented software versioning offers a feasible solution to address challenges posed by frequent and regular software updates. Nonetheless, experiences and feedback from end assembly have revealed limitations, demonstrating that this approach serves as a temporary measure. Based on this feedback, the benefits (marked with "+") and drawbacks (marked with "-") are summarized as follows:

+ *Low development cost*: The proposed software usage of this approach does not require alteration to the in-car software. Instead, it involves deploying a specific, stable software version for the end assembly. Therefore, the



(a) Concurrent development and flashing strategy



(b) Development and flashing strategy with production-specific software versioning

Figure 3.1: Example usage of production-oriented software versioning

corresponding development costs are not raised, with adaptations needed for software management.

+ *Predictable production performance*: Since the software used within the production relies on a consistent software version (e.g., software v_1 shown in Figure 3.1-(b)), the performance of in-car software regarding production-relevant usage is predictable. Namely, the in-line flashing duration and results of commissioning and testing procedures are theoretically dependent of the software version v_1. Therefore, the production performance can be stabilized by using this approach.

- *Limited robustness to regular software updates*: Given that software updates for automotive ECUs are regular and frequent, and software updates can sometimes involve the production-relevant software (Definition 2.1), the scenario arises the software version for production usage (e.g., software v_1 in Figure 3.1) must be updated. This situation leads to usage conflicts. Consider the example that an ECU E_1 whose software for production usage must be updated from v_1 to v_2, cf. Figure 3.1-(b).

  – If the in-line flashing applies software state v_1 for all ECUs while E_1 uses v_2, the compatibility of E_1 to the neighborhood is affected, as the software systems of C_1 and its neighbors are not synchronized regarding inter-ECU communication. As a result, the stability of commissioning and testing of the end assembly cannot be guaranteed anymore.

  – Once the necessity of updating E_1 leads to global software updates for the connected ECU group or even the entire vehicle, the approach becomes impractical, as a stably used software version v_1 cannot be found.

3.1.2 Delta flashing

If for any reason the software for an ECU must be updated, the compiler creates a new binary file for the device. It is common practice that even if the difference between the old and actual binary files are slight, the whole ROM of the µC is reset and installed.

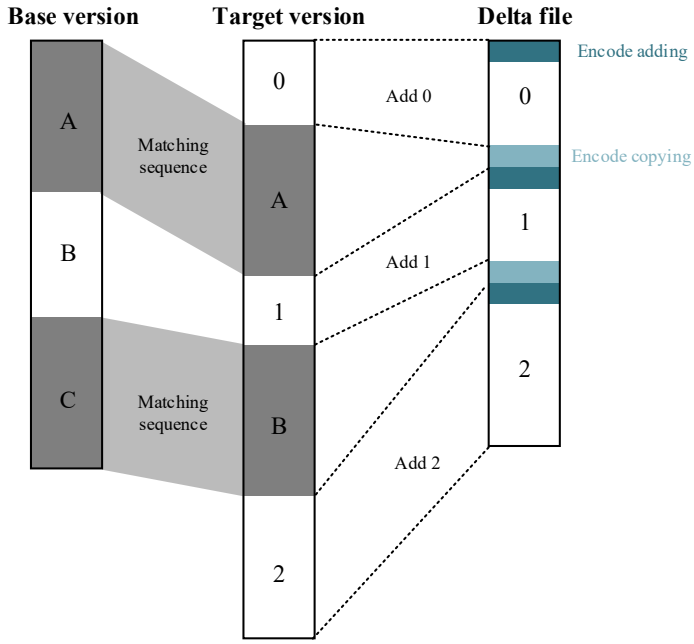


Figure 3.2: Example delta encoding, based on Bogdan et al. [35].

The *delta flashing* solution leverages the slight differences of two software versions, aiming to send only the disparity of the two versions. The disparity description is called delta file, shown in Figure 3.2. The presented delta file encompasses newly added data and encoding commands: adding and copying [35]. Compared to the target software version, the delta file is more compact, which reduces the duration of data transmitting from the source to the target ECU.

In best cases, delta flashing can save up to 90% of software re-flashing [35]. However, delta flashing faces the following constraints and disadvantages:

- *Data depressing*: Delta flashing requires extra time from software development for the data to be decompressed once it is transmitted [35].
- *µC flash structure*: The ROM in most µCs cannot be edited in bit but in page (e.g. 32 Bytes for AURIX TC3xx µC [77]), which forces the difference between two software versions to be centralized to achieve performance of delta flashing.
- *Computing within µC*: The delta file needs to be executed by the target µC that has limited computing power. In case the difference between two software versions is over 10%[1], it is not worthwhile anymore to carry out delta flashing, as the computing time of µC dominates and takes more time than full flash reprogramming.

3.1.3 Shadow memory

If the flash memory of the ECU is sufficiently large or can be expanded, the *Shadow memory* approach becomes an option. This approach divides the program memory into two parts, with only one part storing the active application program. Figure 3.3 illustrates the usage of shadow memory.

In the setup, the memory consists of a bootloader and two sections for the application, with one being active for normal operation.

In case of a software update, the bootloader resets and re-installs only the inactive part of the program memory, while the active part continues to run and provides functionalities. The inactive part, although similar to the active part, keeps unused during normal operation. Hence, it is referred to as *shadow memory*. After the

[1] This threshold is derived from empirical benchmarks conducted in 2020

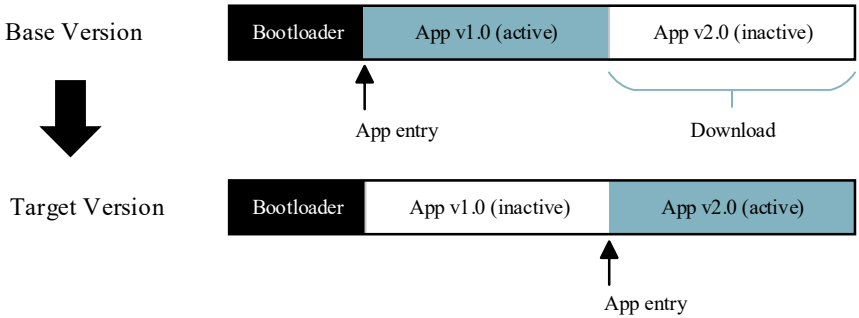


Figure 3.3: Example usage of the shadow memory approach in ROM of µC, based on Lerch et al. [99].

re-flashing, the bootloader changes the application entry, so that the µC executes the new version after restart.

The main advantage of shadow memory is reduction of downtime. This is particularly beneficial for car owners, as software updates can be executed without disrupting usage. For automotive production, the additional memory empowers the usage of frozen software version, maintaining stability in software-relevant procedures during end assembly.

However, the software from two memory parts needs to be configured properly. In case of large software systems, it becomes necessary to handle reset and interrupt addresses [99]. Additionally, the inclusion of extra memory incurs higher costs in terms of µC acquisition. For example, doubling ROM and RAM for a dsPIC controller can lead to a price increase of approximately 70%.

3.1.4 Discussion

The automotive production is increasingly relevant to the in-car software, as highlighted in Section 2.1.2 and Section 2.4.3. Therefore, achieving synchronization between production processes and software upgrade becomes important, particularly regarding regular software updates. Different methods have been explored and utilized to maintain the stability of software-related production process.

However, these methods come with drawbacks, such as high cost (e.g., shadow memory demanding extra flash), tricky technical requirement (e.g., planned memory usage for delta flashing efficiency), or limited usability (e.g. production-oriented versioning only temperately usable). Moreover, these presented approaches, which avoid direct intervention of in-car software, fall short to address the root causes, as the interests of in-car software development and the production usage cannot be separated clearly.

Hence, approaches are desired that directly engage with the in-car software rather than depending on superficial measures. This perspective directs the focus of the thesis towards analyzing the software architecture, which represents the core and structure of in-car software.

3.2 Automotive software architectures

Software architecture serves as the foundation of automotive software design, offering the structures and overviews of the entire software system [145]. When software lacks a well-defined architecture, it often results in disorganized source code that lacks clear roles, responsibilities, and relationships.

Definition 3.1: Software system

A **software system** in automotive embedded systems is a composition of interrelated application Software Components (SW-Cs) and supporting software modules, integrated on designated ECUs to realize specific vehicle functions.

3.2.1 Architectural views and design patterns

Similar to the Vehicle Systems Architecture (VSA) shown in Figure 2.7, automotive software architecture can be considered in different views. The 4+1 architectural

view [91, 145], as shown in Figure 3.4, represents a model for describing the architecture of software-centric systems in the following perspectives:

- *Logical view*: This view entails the design model of the system in perspective of connection, encompassing connectors and interfaces.
- *Process view*: This view involves execution processes of the system, enabling analysis of non-functional attributes of the software.
- *Physical view*: This view handles the allocation and assignment of the SW-Cs onto the hardware platform (e.g., Microcontroller (µC) and High Performance Computer (HPC)).
- *Development view*: This view addresses the organization of SW-Cs in software systems regarding the development process.

Upon the four views described, the *scenario view* describes the interaction of the system with external peripherals, particularly in the context of a use case. In this doctor thesis, the *logical* and *scenario* view are considered in detail.

Considering the logical view of automotive software systems, the essential discussion points are the characteristics of SW-Cs and the relationship between them. In this thesis, a SW-C is conceptualized as entity encompassing logical operations, input and output interfaces, as well as the utilization of hardware peripherals and trigger mechanisms. Based on SW-Cs, the software design pattern is derived as follows:

Definition 3.2: Pattern-based software design

Pattern-based software design is defined as a set of architectural principles guiding the organization of software systems by defining the roles and interactions of SW-Cs within the system [127].

To elaborate, in pattern-based software design, a SW-C follows the principles of design patterns. For instance, a SW-C tasked with abstracting LED functions should not directly control the movement of an electric motor. Each SW-C corresponds a

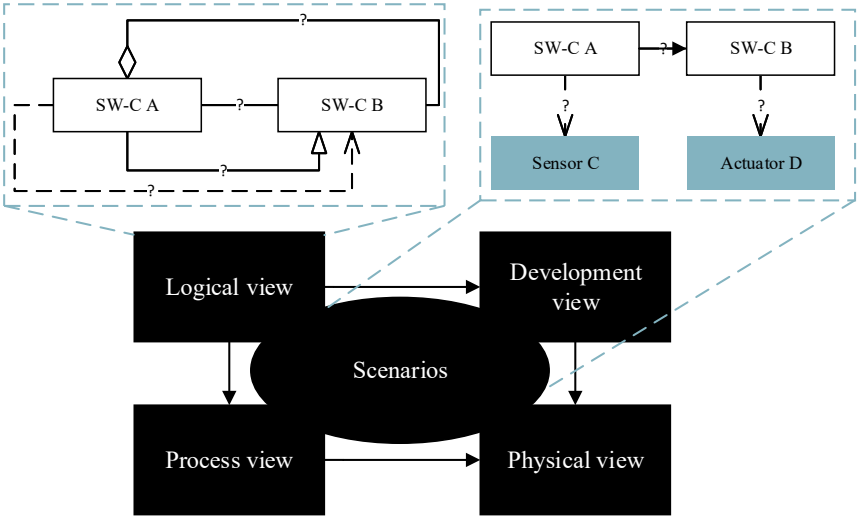


Figure 3.4: 4+1 view model of software architecture, based on Kruchten [91].

role tailored to specific functional requirements, such as implementing LED-related functions and providing application interfaces for other SW-Cs.

Various design patterns can be used to organize a software system. However, only a subset of the design patterns are feasible in the automotive software domain, as automotive software systems have more stringent requirements for reliability and robustness [145]. This section gives a short overview of software design patterns used in the automotive industry.

Layered architecture The layered software architecture, illustrated in Figure 3.5, organizes SW-Cs into horizontal layers with dedicated tasks assigned to each layer [127]. While the application layer implements specific use cases, such as rendering a page on the display, the middleware (Definition 3.3) utilizes hardware-related software to provide interfaces, enabling actions like drawing a button based on parameters. An important aspect of the layered architecture is the call hierarchy between SW-Cs. While the usage of SW-Cs in the same layer

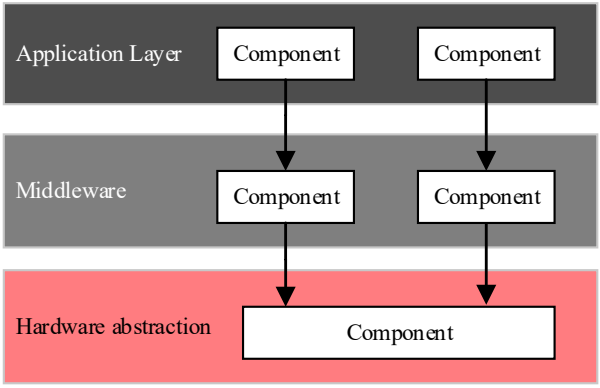


Figure 3.5: Layered software architecture, based on Staron [145].

and the underlining layers can be permitted, the call from lower layers is strictly forbidden in layered architectural pattern.

Definition 3.3: Middleware

In this thesis, the term *middleware* for embedded software systems is defined as the software that reside between the applications and hardware [59]. In embedded systems, the goal of middleware is to integrate reusable SW-Cs to decrease development time [136].

The layered architecture is the de facto standard for many automotive embedded software and Java EE applications [127], serving as a natural choice for separating concerns [54]. In the automotive domain, the AUTOSAR Classic Platform (CP) (see Section 3.3) leverages the layered architecture. In contrast to web server software systems that use databases as the lowest layer, layered automotive software systems rely on a hardware abstraction layer.

Until today, most automotive ECUs exhibit strong interaction with sensors, actuators, and setpoint generators. The limited memory and computing power of the µCs within the ECUs necessitate leveraging restricted resources to ensure functionality and cost discipline. In this context, the layered architecture in automotive software

provides the feature of *software-hardware reflection*, ensuring the separation of concerns and simplifying the localization of software functionalities.

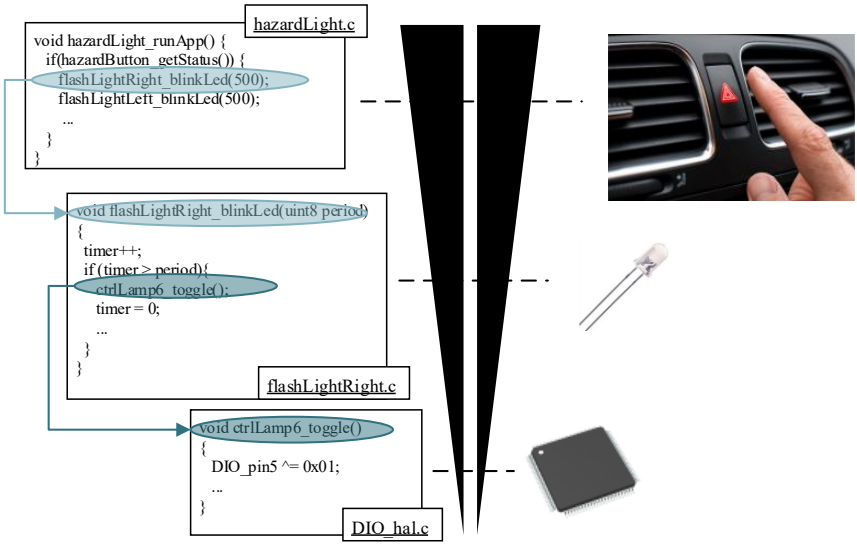


Figure 3.6: Example codes for vehicular hazard light function, demonstrating hardware-software reflection of layered architecture.

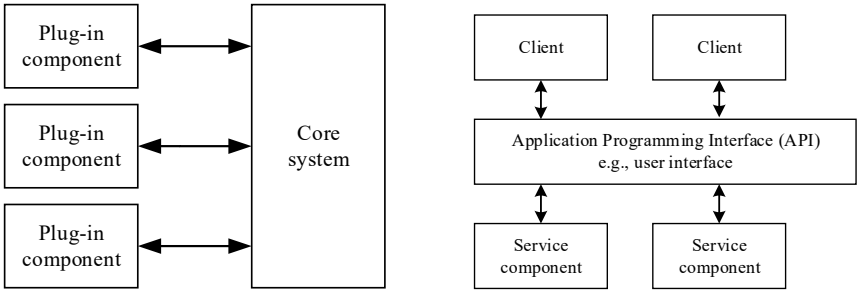
Figure 3.6 presents an example of software-hardware reflection regarding the vehicular hazard light function. The principle of the function is that the pressed hazard light button on the head unit triggers the blinking of a hazard light LED with given period. Examining the in-car software with regards to layered architecture, the SW-Cs can be structured into a three-layer hierarchy:

- *Hardware abstraction layer (HAL)*: The file *DIO_hal.c* represents the abstraction of hardware on the board of µC. Specifically, the lowest layer of automotive embedded software links the interfaces of µC and the corresponding peripherals. In this show case, the pin 5 of Digital inputs and outputs (DIO) is linked to Lamp 6 of the E/E system.

- *Peripheral abstraction*: The peripheral abstraction layer upon HAL is a reflection of the sensor, actuator, or setpoint generator. This layer is dedicated to build the performance and features of connected hardware that is not located on the board of MCU. In context of the hazard light, the peripheral abstraction is given in the file *flashLightRight.c* and reflects the behaviors of the LED, e.g., blinking.

- *Application:* The functionality of the hazard light is built upon cooperation with other peripherals. In this context, the hazard light LED interacts with the hazard light button. Therefore, the file *hazardLight.c* representing the Application Layer realizes the desired function based on the abstraction of MCU hardware and peripherals.

In the automotive domain, the widely used AUTOSAR Classic Platform (CP) [22] leverages a similar principle to the demonstrated software-hardware reflection. While the example shown in Figure 3.6 realizes the hazard light function with hand-written C code, CP methodology [23] abstracts the MCU hardware by configuration options and provides code templates for the development of peripheral-related behaviors and software applications. The CP is discussed in detail in Section 3.3.1.



(a) Microkernel architectural design pattern, based on Richards [127]

(b) Service-oriented architectural design pattern, based on Richards [127]

Figure 3.7: The microkernel and service-oriented design patterns

Microkernel architecture The microkernel architecture, presented in Figure 3.7-(a), comprises a core system and several plug-in components. The core system typically works as an operating system, where only fundamental functionalities are provided. In this architecture, application logic is distributed independently among these plug-in components, offering flexibility and extensibility [127].

The core system executes the minimal essential functions necessary to interface with the underlying hardware. Applications and drivers are subsequently plugged into the core system and run within their designated spaces [34].

In the automotive domain, the operating system QNX [33] adopts the microkernel architecture, providing reliability, performance, and real-time capability to support multiple Advanced Driver-Assistance Systems (ADAS) functions. In return, high-end processors are necessary to drive this microkernel-based software platform [124].

Service-oriented architectures In modern software engineering and automotive software domain, the terms *service-oriented architecture*, *service-based architecture*, and *microservice architecture* are widely spread. These architectures differ in their considered scope [71] and granularity of service components [6], but they all emphasize the use of services as fundamental design patterns. This thesis treats service-oriented architecture, service-based architecture, and microservice architecture as synonymous terms.

Definition 3.4: Services in software systems

Services are defined as crosscutting interaction aspects within software systems, factoring out collaboration among the SW-Cs to fulfill a certain task [39].

Figure 3.7-(b) illustrates the principle of service-oriented architecture. Based on a commonly used application programming interface (API), different service components can be independently deployed, allowing decoupled realization of business applications. The API enables the client requests to be handled by

deployed services. Interactions between service components (consumption) allow the system to deliver applications that involve multiple service components.

In the automotive domain, the AUTOSAR Adaptive platform [17] adopts a service-oriented architecture leveraging Internet-based protocols [145]. While not yet widespread in the automotive industry, service-oriented architecture exhibits potential in addressing the challenges posed by frequently changing automotive software.

This section delves into various established software architectures and analyses their characteristics, serving as a foundation and background for the subsequent exploration of the specific thematic discussion regarding embedded software usage in the automotive industry.

3.2.2 Signal- and service-oriented architectures

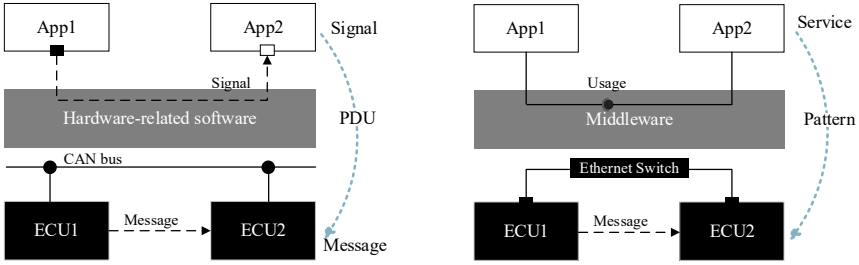
Two types of architectural styles are derived from communication methods between application SW-Cs in automotive software: signal- and service orientation.

Definition 3.5: Signal-oriented architecture

Signal-oriented architecture describes software systems in which information exchange between SW-Cs occurs through signals transmitted from senders to receivers [114].

With the usage of services, explained in Definition 3.4, the service-oriented architecture in the field of automotive software leverages the following two communication principles [94]:

1. *Request / Response*: For instance, a determined action is called and an acknowledge is returned, and
2. *Publish / Subscribe*: For example, a client subscribes to a service provided by a server and gets notified periodically.



(a) Mapping in a signal-oriented architecture

(b) Mapping in a service-oriented architecture

Figure 3.8: The principles of signal to message mapping in signal- and service-oriented architectures, based on Kugele et al. [94].

Figure 3.8 illustrates the paradigm shift between signal- and service-oriented architecture. In a signal-oriented environment, such as illustrated in Figure 3.8-(a), where two applications (*App1* and *App2*) on separate ECUs exchange information, the signal leverages Protocol Data Units (PDUs) to translate into messages on a specific bus (e.g., the CAN bus). The translation via PDU, regulated by a predefined table, or the communication matrix [176], demands that the development of application software align with this communication matrix. This characteristic results in the inflexibility of the signal-oriented architecture, as any alteration to the communication matrix necessitates updates in the corresponding application software, requiring a full recompilation of the embedded software. Moreover, this communication method does not support any dynamic addition of new hardware or functions at runtime [135].

In contrast, in the service-oriented architecture, only interaction patterns are specified, as depicted in Figure 3.8-(b). The interaction between application software occurs dynamically at runtime through the Ethernet backbone. Dedicated middleware orchestrates the dynamic communication between applications, facilitating the decoupling of software and hardware. The utilization of service-oriented architecture enables dynamic communication and reconfiguration of network topology during runtime [94].

Regarding the development process, the partial communication matrices for ECUs must be defined within signal-oriented systems, before the software development starts. Therefore, the development of signal-oriented architecture is ECU-based.

Table 3.1 summarizes the differences between signal- and service-oriented architectures in several aspects.

Table 3.1: Comparison of signal- and service-oriented architectures

Aspect	**Signal-oriented**	**Service-oriented**
Artifact	Signal, PDU, communication matrix	Service, client
Development	ECU-based	Service-based
Deployment	Static	Dynamic
Communication bus	CAN, FlexRay, LIN	Ethernet
Hardware-related software	Linking the applications and hardware	Orchestration of service consumption
Usage in vehicles	Low-level ECUs, hard real-time demands	Centralized ECUs, soft real-time demands

3.3 AUTOSAR platforms

AUTomotive Open System Architecture (AUTOSAR) stands out as one of the most extensively adopted architectural standards for automotive software systems [11]. Established in 2003, the AUTOSAR consortium, encompassing OEMs, suppliers, and other industry stakeholders, has been working on the development and introduction of a standardized software architecture for the automotive industry.

The key idea of AUTOSAR architecture is standardization and decoupling of software and hardware. Varying in the field of application, there are two variants of AUTOSAR:

- **AUTOSAR Classic Platform (CP)**: This platform is *signal-oriented* and serves as a solution for embedded systems characterized by hard real-time requirements and safety constraints [25].
- **AUTOSAR Adaptive Platform (AP)**: The adaptive platform is *service-oriented* and is designed as a solution for HPCs, enabling the construction of safety-related systems for applications such as highly automated and autonomous driving [25].

The subsequent paragraphs delve into the technical details of the two AUTOSAR platforms. Following the introduction, this section briefly explains why layered and signal-oriented architecture will be further used in the automotive industry.

3.3.1 AUTOSAR Classic platform

The AUTOSAR Classic Platform (CP) leverages a layered software architecture using signal-based communication. Illustrated in Figure 3.9, the CP consists of three layers built upon the µC: the Basic Software (BSW), encompassing system services, communication services, I/O drivers, and more; the Run Time Environment (RTE); and the Application Layer.

The primary goal of the CP is to realize the decoupling of software and hardware, fostering cooperation among various stakeholders in the automotive software development, including OEMs, Tier 1, and suppliers. Typically, OEMs offer the communication matrix and function requirements, based on which Tier 1 develops SW-Cs in the Application Layer. Simultaneously, the suppliers work on deriving the suitable hardware (µC, sensors, and actuators) and configure the BSW. The integration of BSW and SW-Cs is achieved using the Run Time Environment (RTE), resulting in functional software packages.

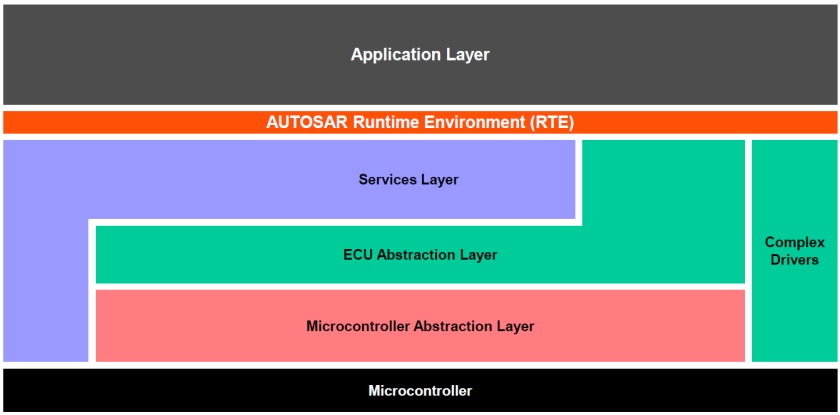


Figure 3.9: Overview of software layers in the AUTOSAR Classic Platform (CP), based on AUTOSAR consortium [22].

A more detailed introduction to AUTOSAR Classic Platform is given in Section A.2.

As the volume of software within automotive ECUs continues to expand, achieving an ideal layered cut, as illustrated in Figure 3.6, becomes progressively challenging. This is due to the escalating dependencies between various SW-Cs. In case that the AUTOSAR platform was not used, the programmers would tend to use the most direct way to realize desired functions, which causes the erosion of software-hardware decoupling.

3.3.2 AUTOSAR Adaptive platform

The AUTOSAR Adaptive Platform (AP) is an emerging standard designed to address the needs of highly parallelized and distributed systems, such as realization of autonomous driving [10]. The AP follows a service-oriented architecture that the application usage is dependent of the needs, see also Section 3.2.2.

Figure 3.10 illustrates the logical view of the AP. The AP separates functional applications and a Portable Operating System Interface (POSIX)-compliant operating system through a modular layer known as the AUTOSAR Runtime for Adaptive

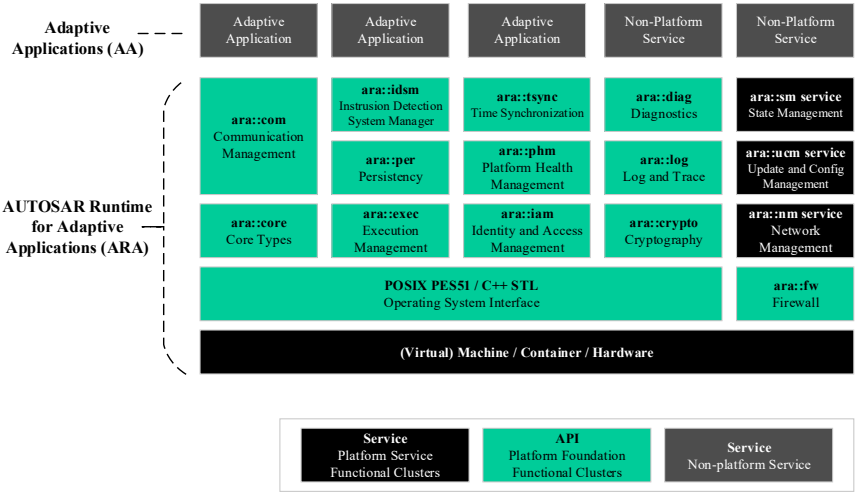


Figure 3.10: Architecture overview of the AUTOSAR Adaptive Platform (AP), based on AUTOSAR [17].

Applications (ARA). The ARA defines interfaces and services to standardize the access of operating system and communication channels, which enables the realization of a service-oriented architecture based on Ethernet [66].

Comparatively, the ARA can be analogized to BSW of the CP, as both BSW and ARA are based on a (virtual) machine and support the application software. While the positions of BSW modules indicate the dependencies and hierarchy, the Application Interfaces (APIs) upon the POSIX operating system are of the same value and partially independent.

The AP employs various proven technologies not fully used by ECUs in the CP. Notably, key technology drivers of the AP include the usage of Ethernet and manycore processors. Although the CP supports Ethernet, it is primarily designed for the legacy communication technologies, making it difficult to fully utilize and benefit from the capability of Ethernet. Additionally, the CP supports usage of multicore processors as well, but it falls short in achieving high computing efficiency offered by parallel computing of GPU, FPGA, and accelerators [21].

The usage of the service-oriented architecture and the POSIX operating system provides the AP flexibility and scalability in processing distribution and computing power allocation, which in further, support high-speed OTA software update. Consequently, a transition from ECU-based development to service-oriented development is proposed as a means to address the complexity in the development process of the automotive software [86, 93, 164].

3.3.3 Reasons for retaining the CP

Despite the improvements introduced by AP, a great part of automotive software will still be designed and implemented in layered and signal-oriented architectures like the CP. The durability of the CP can be attributed to several factors:

- **Real-time capability:** While AP can ensure low latency down to milliseconds through technologies like Ethernet Time Sensitive Networking (TSN) [40], applications with hard real-time requirements, such as motor control, demand a real-time implementation with hardware-related configurations (e.g., MCU interrupt mechanisms). Consequently, AP may not be suitable for these applications.
- **Cost:** The usage of AP leads to the cost of HPC, as computing power is required for the POSIX operating system and virtualization. For specific ECUs with limited improvement possibilities, like door modules, where regular MCUs cost only a few Euros, the increase in cost for HPC is financially impractical.
- **Wiring harnesses:** The adoption of HPC implies centralization of computing power and peripherals. While the centralized E/E architecture with HPC usage optimizes the number of ECUs, connecting existing peripherals to HPCs can be challenging due to wiring harnesses.

- **Existing development:** Validated ECUs in recent car series have undergone thorough validation during the development process. Repeating the development for the same functions on new platforms is not financially desirable for both OEMs and suppliers.
- **Hardware-related interface:** The AP uses Ethernet exclusively as communication methods, causing the usage of peripherals with other communication protocols such as UART, IIC and SPI. An API must be built if there is no existing gateway for those peripherals.

Therefore, this thesis states that the CP and the AP will coexist in automotive ECUs in the future, demanding organic cooperation between signal- and service-oriented systems. Especially, the software design on the CP should take more factors into consideration. On the one hand, CP-based ECUs must maintain performance to meet real-time requirements and offer corresponding interfaces for dedicated peripherals. On the other hand, these ECUs must provide flexibility in configurations and interfaces that align with the usage of AP which is deployed on their upper machines.

Regarding the automotive production process, CP-based ECUs should be considered with more severity, because these ECUs are directly connected to sensors and actuators, necessitating corresponding testing procedures during the end assembly. Additionally, the µCs leveraging the CP have limited computing power and a slower communication channel than Ethernet, leading the in-line flashing process for those ECUs more challenging than those using the HPC. Therefore, this thesis focuses on architectural approaches and methodologies for signal-oriented systems like the CP.

3.3.4 Model-based development within AUTOSAR

Model-Based Systems Engineering (MBSE) is formally defined as the application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing

throughout development and later life cycle phases [73]. Within the development reference model shown in Figure 2.6, System Architectural Design (SYS. 3) and Software Architectural Design (SWE. 2) can be carried out with model-based approaches, leading the following development procedures (SWE. 1 and SWE. 3) oriented by the created models. Additionally, the system integration and tests, e.g., SWE. 5 and SYS. 4, can be designed and executed with help of the established models.

In the automotive domain, MBSE is increasingly popular in the development of embedded software systems. The model-based approach for automotive software development enables evolutionary and platform-independent development of automotive systems, reducing development times and improving product quality [137].

Modeling Automotive Software

MBSE is used to support analysis, design, and verification of the developed systems as part of the systems engineering process [55]. Regarding the software development, the models are used to describe requirements, behaviors, structures, or parameters of software systems. These models are typically expressed using specialized modeling languages. An Architecture Description Language (ADL) allows for a formal representation of architectures, enabling semantically accurate architecture descriptions [111]. Automotive-related ADLs encompass EAST-ADL [47], AADL [52] (adapted from avionics), SysML [121], and architectural views extracted from these ADLs [43].

Additionally, SysML is evaluated as suitable for modeling automotive software systems [43]. SysML is a general-purpose system architecture modeling language for systems engineering application [121] and includes nine diagram types as shown in Figure 3.11.

In this thesis, the Block Definition Diagram (BDD) and Internal Block Diagram (IBD) of SysML (the green blocks in Figure 3.11) are employed to describe software systems and sub-systems, as the emphasis of the thesis lies in exploring the structure of embedded software within µCs, rather than the realization methods

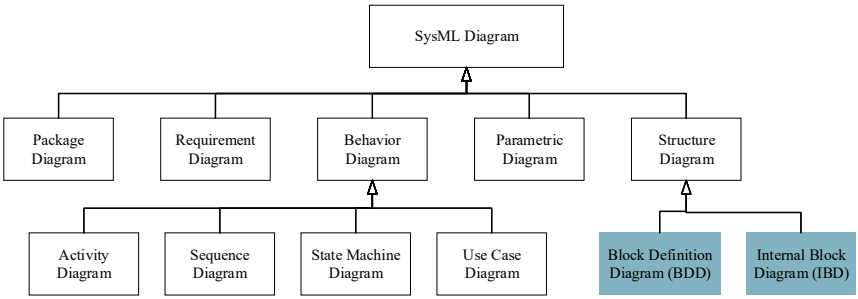


Figure 3.11: SysML diagram taxonomy, based on Friedenthal et al. [55].

of SW-Cs. Both BDD and IBD are modified UML class diagrams, with BDD representing structural elements, the composition and classification, while IBD represents interconnections and interfaces between parts of a block [55].

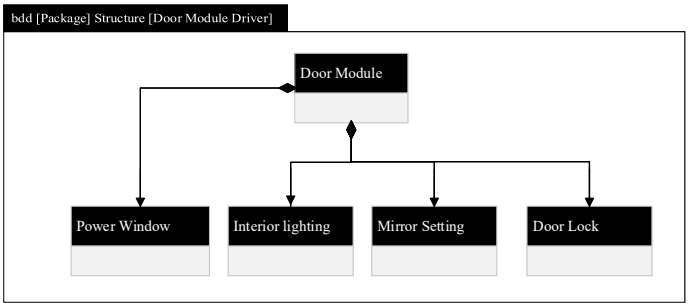


Figure 3.12: An example Block Definition Diagram (BDD) for the Door Module Driver (DMD) in the car

From a systems engineering perspective, BDD serves to describe the hierarchy of a system, similar to a part tree. The blocks within BDD can be refined using IBD which shows the interfaces of the sub-systems and interconnection among them.

Taking the example of power window function in the vehicle, Figure 3.12 illustrates the structural view of the belonged ECU, the Door Module Driver (DMD). The DMD is responsible for interior lighting, mirror setting, door lock, power window,

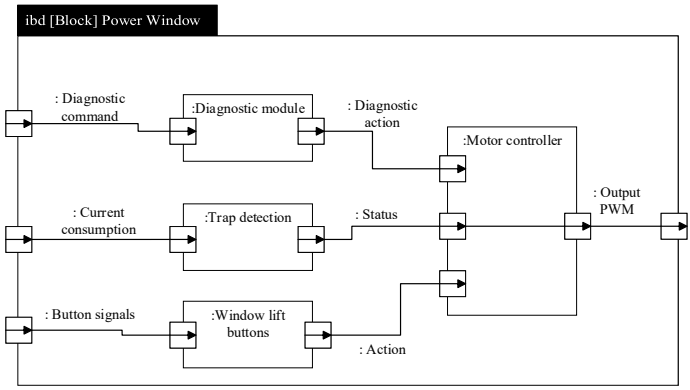


Figure 3.13: An example Internal Block Diagram (IBD) for the power window functions in the Door Module Driver (DMD).

etc. [69]. The BDD presents the relationship between the system *Door Module* and its functionalities.

Figure 3.13 follows the BDD of DMD, demonstrating the software structure for the power window function using the IBD. Within the IBD, logical function modules and interfaces between modules are illustrated. Specifically, the vehicular power window uses diagnostic commands, current consumption, and button signals as inputs to generate output for the window motor. The SW-Cs, e.g., *Diagnostic module*, with the interfaces are demonstrated in the IBD.

Definition 3.6: Model-based and model-driven processes

Model-based processes use models to describe requirements and designs, serving as references throughout the development process (see also Section 2.2). Changes of model entities in model-based processes do not automatically cause changes to the software system.

Model-driven processes rely on models where model entities directly generate code and have a direct impact on the software.

Model-driven development within the AUTOSAR Classic platform

AUTOSAR is a standard for component-based software design and thus not an Architecture Description Language (ADL) [43]. The methodology of the CP describes a rough outline of the design steps to build a executable software for µCs [23]. However, the CP does not define roles, responsibilities, and usage of model-based engineering. In theory, model-based engineering utilizes models as references, documentation, or guidelines. However, the thesis focuses on model-driven processes that directly influence the software development process according to Definition 3.6.

Figure 3.14 showcases a development workflow for an µC-based ECU, involving three stakeholders: the OEM, Tier 1s, and suppliers. This workflow corresponds to the development process SWE.1, SWE.2 and SWE. 3 in Figure 2.6.

Following the Requirement, Functional, Logical, and Physical (RFLP) perspectives introduced in Section 2.2, and assuming that an desired vehicular function require cooperation among several ECUs, the development process starts with the formulation of requirements (Block 1 in Figure 3.14). The OEM continues with defining user stories (Block 2), scoping the functional perspective of the desired vehicular function. Subsequently, the OEM specifies which signals used in the software system. The names, meaning, and position in payloads of these signals are documented in the communication matrix (C matrix) for the relevant

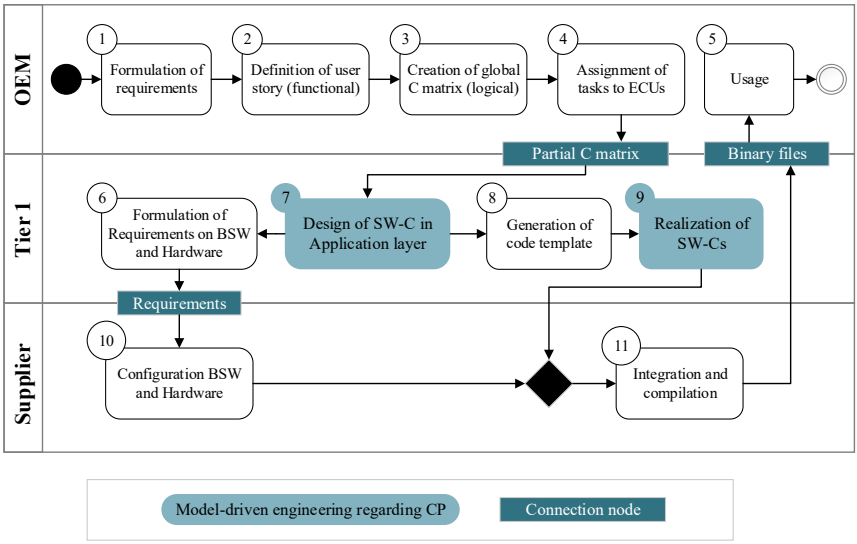


Figure 3.14: Example development process of ECU software regarding the AUTOSAR Classic Platform (CP)

bus system. The creation of global C matrix (shown in Block 3) addresses the interfaces between different software sub-systems, thereby covering the logical perspective. Following the global C matrix, the OEM should assign tasks to dedicated ECUs (Block 4), leading to the derivation of partial C matrices for each ECU. These partial C matrices are further refined by Tier 1.

The development processes described above are primarily carried out by OEMs and are predominantly document-based. The model-driven development in automotive software development domain comes into play when the Tier 1s use partial C matrix to formulate SW-Cs within the Application Layer (Block 7 in Figure 3.14), see also Section 3.3.1. These SW-C can be conceptualized as model components with interfaces.

The architectural design of SW-Cs can be executed using model-driven development tools like DaVinci Developer from Vector Informatik [159] or EB tresos Studio from Elektrobit [49], as shown in Figure 3.15. Inside of each SW-C, several

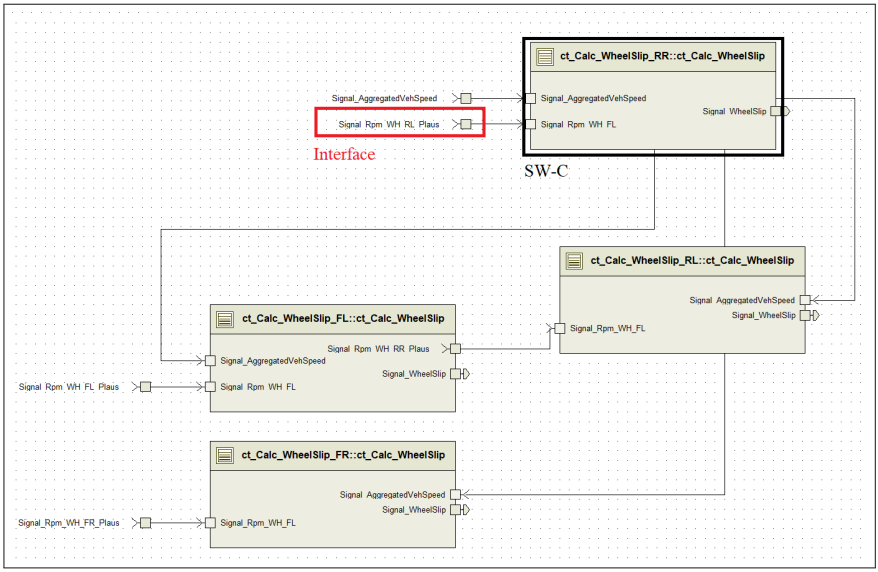


Figure 3.15: Model-driven development tool for SW-Cs of the CP within DaVinci Developer of Vector Informatik [159].

independent task entities (referred to as Runnables in AUTOSAR terminology) can be created, implementing algorithms to realize desired functions of the SW-C. Each task entity corresponds to a function in programming language C.

Definition 3.7: Compilation and compiler

Programming languages, such as C for automotive embedded software systems, are notations used to describe computations for both humans and machines. However, to execute a program on a machine, it must first be translated into a machine-executable format tailored to the target platform, such as an µC-based ECU [3]. This translation process is called *compilation*, and the software that performs this translation is referred to as a *compiler*.

Once the software architecture of the ECU is established, requirements on BSW and hardware can be derived (Block 6). Suppliers can then configure BSW and determine the utilized hardware based on the requirement (Block 10).

With the fully configured BSW and the finalized design for SW-Cs, code templates can be generated (Block 8), within which function developers implement the algorithms. The realization of SW-Cs (Block 9) can also be supported by a model-driven process. For instance, C code filling the code template can be generated from Simulink models of Mathworks [156].

For µC-based automotive software systems, binary files (e.g., .hex files) are generated from the source code and configurations from the *compilation* process (see also Definition 3.7, Block 11 of Figure 3.14). These binary files are then deployed to µC-based ECUs for execution (Block 5).

Discussion

Adopting MBSE in automotive software development results in benefits, ranging from reduction of development time to overall design improvement, model re-usability and functional attainability with enhanced product quality [150].

Within the development process regulated by CP methodology, roles and interfaces can be defined. For instance, the use of communication matrices, requirements regarding BSW modules, and the generated code templates establish separation of each phase during automotive software development. In the process, software

architects outline the global communication matrices, functional architects derive the local communication matrices for ECUs, and functionalities within the ECUs are realized with the given signals. Therefore, the use of model-driven development within the CP supports the separation of concerns and enables seamless verification of software functionalities up to the first software release, as the results of each development phase are clearly defined.

Compared to other object-oriented programming languages, C and C++ are *"sharp knives without handles."* This is because every function programmed by any developer can change hardware registers, causing unexpected global effects. Against the background, the shown model-driven development process restricts the power of function developers and software developers by defining the input and output signals of each SW-C. This ensures that local errors within SW-Cs do not significantly affect global performance.

However, the illustrated process shown in Figure 3.14 faces challenges regarding regular software updates and emergence of new functionalities:

- The demonstrated process does not fully adhere to the principle of the RFLP perspective. The communication matrices which are fully defined in step 3 in Figure 3.14 address both logical and physical perspectives, as the names, meanings, and layouts of signals imply the realization methods of desired function. The determination of signals restricts the functional design of Tier 1 which is shown in Block 6 in Figure 3.14.
- This process is fragile concerning regular software updates. Software updates can be corrective, adaptive, or perfective [60], causing changes of used signals. Within the illustrated development process, every signal update lead the development back to alteration of global communication matrix (Block 3 in Figure 3.14). Therefore, software updates demand seamless cooperation of OEMs, Tier 1s, and suppliers, which can be challenging after the Start of Production (SOP).

3.4 Software refactoring

The term *software refactoring* was firstly introduced by the PhD thesis of Opdyke [118], where it referred to restructuring within object-oriented languages. However, the concept of software refactoring gradually evolved to encompass general software evolution aimed to improve software quality.

Definition 3.8: Software refactoring

Software refactoring is defined as a technique that alters the internal structure within the software without changing its external behaviors by transforming software artifacts [1].

Adapting a production-optimized software architecture into existing automotive software systems requires unchanged customer functions, which corresponds the definition of software refactoring. Additionally, the objective of software refactoring is to improve the software quality including extensibility, modularity, reusability, and maintainability [103]. Therefore, this section introduces and analyses software refactoring methods for automotive software systems.

In essence, software refactoring involves methods aimed at enhancing desired quality metrics of software (see also Section A.3).

Software refactoring involves sequential activities to ensure the results. An approach to implement software refactoring can be summarized as follows [29, 103]:

1. Identification of the software to be refactored,
2. Determination of refactoring methods,
3. Assurance of consistency regarding external behaviors of software,
4. Application of software refactoring,
5. Assessment of refactoring on quality characteristics, and
6. Commitment of interfaces to other software artifacts.

Software refactoring is useful in some cases, such as enhancement of software reusability [109], but the concrete approaches vary according to software programs. Most techniques regarding software refactoring address software systems within object-oriented environments, as the relationship between objects (classes) gets obscure with software evolution. Common approaches of software refactoring for object-oriented languages contain the following manners: composition of methods, relocation of features among objects, simplification of conditional expression, generalization of classes, data organization, simplification of method calls, and mixed refactoring [1].

For software systems building on object-oriented programming languages, test-driven development [51] and data-oriented programming [139] are proposed to deal with legacy code, reduce the complexity of software systems, and increase the flexibility. In test-driven development, failing test cases are the prerequisites of compiling function codes, which allows writing new code independently of old code. The data-driven programming makes two independent parts of a software system: one involves only data entities, and the other involves only code entities. However, dealing with the legacy code in software systems with process-oriented languages (such as C) is still challenging [51].

Given that CP-based software systems are implemented in C, refactoring them is challenging. Nevertheless, refactoring within the CP is still feasible and can be approached in two categories, as discussed below:

- **Refactoring in BSW**: As the CP regulates the layered software architecture and the methodology rather than the concrete implementation, local improvement is feasible regarding BSW modules. For instance, Galla et al. [58] apply a component-based paradigm to refactor the modules FlexRay Driver and FlexRay Interface, which reduces memory consumption and execution time.
- **Refactoring in Application Layer**: Another perspective is to bring discipline into the Application Layer. As SW-Cs are built upon BSW, logical analysis of SW-Cs regarding the function requirements can improve

reusability and modularity of considered software systems. Vogelsang [168] analyzes the dependencies between features in automotive software systems and introduces a dependency-based refactoring approach to extract feature dependencies between SW-Cs. The study on feature dependency indicates the lack of architectural discipline within the Application Layer and suggests additional layers in automotive software systems to achieve modularity. Additionally, Lee et al. [97] propose algorithms to isolate Runnables with different criticalities temporally and spatially within the CP.

In general, software refactoring in automotive field is more challenging than the others, though automotive software requires regular updates nowadays. The reason can be summarized into two main backgrounds:

- **Process-oriented programming**: Until today, automotive software is usually implemented in embedded systems with process-oriented language C, as the development process in Figure 3.14 illustrates. Nevertheless, process-oriented languages are especially challenging in a legacy environment, as usage of unit tests in process-oriented environments is severely limited [51]. Taking the µC-based ECU as example, the common practice is changing source code, flashing into the µC, and hoping the changes were right. The approach of global tests to validate local changes makes software refactoring inefficient.
- **Model-driven development**: Although the AUTOSAR Adaptive Platform (AP) leverages C++ instead of purely process-oriented C, the function development can keep model-driven. However, model-driven development is also recognized as rigid regarding software redesign [137]. Furthermore, it is a common practice in the automotive industry to generate process-oriented code (C or non-class C++) with tools such as TargetLink [46] and Simulink Coder [156]. The close relation between model-driven development and process-oriented languages results in general difficulty of refactoring automotive software.

From a broader perspective, software refactoring is a compulsory approach in large and evolving software systems, as functional increase always brings corresponding quality declining and complexity increment [98]. Therefore, software systems must be maintained on time to accommodate the new requirements. The lack of regular software refactoring can result in accumulating *technical debt*, which has to be paid back with higher cost in the future [92].

3.5 Potentials of production-optimized software architecture

In view of automotive manufacturing, the automotive software operates as a black box, with delivered binary files being monolithic and non-explanatory. Interfaces such as diagnostic command lists are pre-defined for commissioning and testing procedures. In case something goes wrong, end assembly technical staff are not able to analyze and resolve the problems, forcing the delivered software to be mature and stable.

In the view of automotive software development, the rigorous development methodology of the CP has evolved over 20 years and contributed to the quality of the whole vehicle. Although the CP considers modularity, reusability, and separation of concerns within its design, today's automotive software poses challenges regarding the usage after the SOP.

3.5.1 Challenges of automotive software

Against the background of software usage within automotive end assembly line (see Section 2.1.2) and the development methodology of the CP (see also Section 3.3.4), two main challenges can be derived: in-line flashing duration and application dependent diagnostic interface.

In-line flashing duration

The discussion on in-line flashing is addressed in Section 2.4.2, illustrated by Figure 2.16. While in-line flashing offers OEMs flexibility by enabling last-minute software updates, it challenges the assembly line by necessitating external testing devices and power supply.

In the current landscape, OEMs are dedicated to increasing the update frequency in order to deliver vehicle features to customers in time. The update frequency can occur as frequently as every four weeks [113], while the global delivery of ECUs may take up to 16 weeks [143]. As a result, OEMs must employ in-line flashing to guarantee the newly produced cars have the latest software. However, software update usually leads to larger binary files to be installed, potentially extending in-line flashing duration.

The in-line flashing process can also be influenced by the dynamic behaviors of related ECUs. Moreover, the flashing tool in the development department differs from the tool in the end assembly area. As a result, predicting the actual time consumption to flash a new software version is tricky because the testing device in the production differs from the development tools and the ECUs during assembly are situated in different E/E topology. The unpredictable duration makes the optimization of flashing sequences even more challenging.

A single instance of overtime during in-line flashing can result in a produced car incapable of being driven off the assembly line, see also Figure 2.16. Given that an end assembly line can manufacture vehicles at a rate of up to one car per minute [85], and considering that a factory may encompass multiple assembly lines, continuous failure in managing the duration of in-line flashing can result in a situation where vast parking slots are filled with over 1000 undrivable cars. These vehicles would then need to undergo manual re-flashing, commissioning, and testing processes.

Such an issue can escalate to the point of need of rebuilding and reconstruction of the assembly line. Besides the high cost of assembly line reconstruction, some assembly halls may not be extended spatially.

Therefore, the quantity of automotive software should be considered not only from the perspective of automotive manufacturing but also in the design and development phases of automotive software.

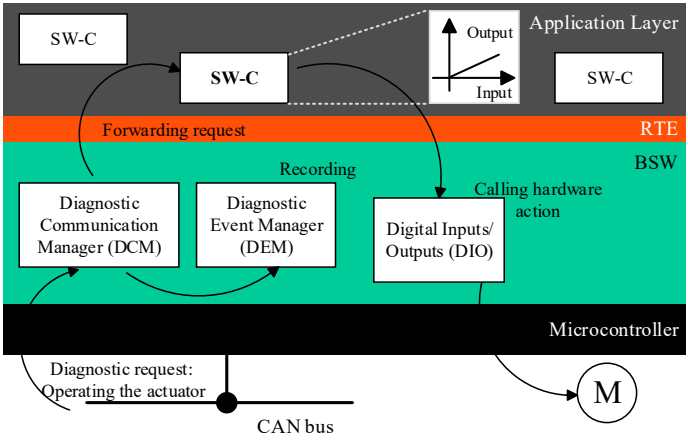
Application-dependent diagnostic interface

Another topic of software usage is more structural and can be derived from the utilization of the CP. The handling of diagnostic requests concerning hardware components, such as sensors and actuators, within the CP relies on SW-Cs in the Application Layer. This characteristic is referred to as *application-dependent diagnostic interface* [Z5].

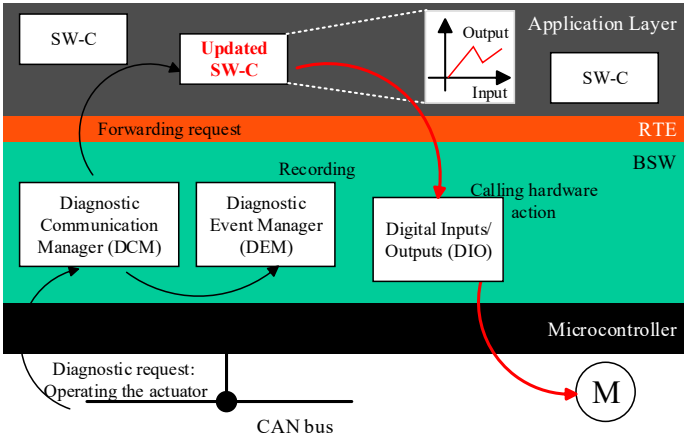
In Figure 3.16-(a), the process of testing the functionality of an ECU regarding an actuator (e.g., an electric motor, denoted by the *M* symbol in the figure) is demonstrated. This process is diagnostic-based and similar to the calibration process shown in Figure 2.17.

The testing process starts with the ECU receiving the diagnostic request from the communication bus (e.g., CAN bus in the figure). Within CP-based ECUs, the diagnostic request is then processed by BSW, where the module Diagnostic Communication Manager (DCM) is responsible for diagnostic data flow and diagnostic state management [14]. The Diagnostic Service Processing (DSP) submodule in DCM analyses the request message, acquires data and executes the required function call on the Diagnostic Event Manager (DEM) module and SW-Cs in the Application Layer [13]. As functions of DEM are insufficient to process the request, the diagnostic function is called in the corresponding SW-C. The SW-C then accesses the digital inputs/outputs module in BSW to control the electric motor.

If for any reasons, the SW-C mentioned above is updated, see Fig. 3.16-(b). A new algorithm is introduced to enhance stability or efficiency. Now a new control algorithm dominates the motor. The same diagnostic request is sent, but the measured output from the motor is affected by the newly introduced algorithm. The new software causes a commissioning failure. Since handling diagnostic



(a) Processing of diagnostic request within CP: functional test of an actuator



(b) Processing of the actuator test with a changed SW-C within the CP

Figure 3.16: Illustration of the application-dependent diagnostic interface

requests for hardware and hardware-related functions can rely on the SW-C, the above-described scenario is named *application-dependent diagnostic interface*.

The failed testing process can be interpreted as hardware damage in the end assembly. It is impossible for the installation staff to identify the real reason for such an accident in the end assembly area. Therefore, systematic approaches towards automotive software and software architecture are desired to address the challenge.

One may argue that the challenge can be attributed to organization and process as a new version of the software must always be published with a new rule for commissioning and testing. In practice, the synchronization of software and corresponding rules is also challenging as the synchronization is a manual process including updates of textual requirement documents. Besides the reason mentioned above that the development engineers might not know the manufacturing process, involving human work in software update release may be risky and even bring uncertainty to the production line.

3.5.2 Research gaps

The sections above present the life cycle of automotive software, tracing its journey from architectural design to its integration into the end assembly process. Especially, the influence of software architecture on both the development and production processes is highlighted. The software related issues, *in-line flashing duration* and *application-dependent diagnostic interfaces*, pose challenges to the efficiency and stability of the value-added process during the end assembly.

The impact can be traced back on the technical detailed design and methodology of the AUTOSAR platforms, which have evolved about two decades. While the established architecture must adapt to evolving requirements, including the increasing frequency of software updates, there remains disparities in understanding regarding the software-related usage in automotive production. The disconnected understanding comes from a lack of familiarity with end assembly processes

among developers and shortage of software development knowledge among production planners. Furthermore, bridging common understandings between staff from production and development teams proves challenging, compounded by organizational separations, as illustrated in Figure 3.14.

Consequently, there is a gap in the development of production-optimized automotive software today, with few metrics available to evaluate software usage during the manufacturing. Moreover, the strong interaction between automotive software and hardware, coupled with factors like cost constraints and real-time requirements, necessitates the use of model-driven approaches with signal-oriented layered architectures. However, the usage of model-driven approaches and signal-oriented systems often results in rigidness regarding software refactoring, making it difficult to improve existing software towards a production-oriented focus.

From a broader perspective, every architectural decision in the early phase of development can incur corresponding impact on the later usage. Therefore, a product that evolves over time must have the agility to adapt to user feedback and changing requirements. However, the current landscape of automotive software architecture falls short in this regard, struggling to strike a balance between executing frequent software update and providing a stable environment for production-related processes.

4 Concepts of Production-Optimized Software Architecture

As discussed in Chapter 1, the requirements on automotive software are multifaceted, from which a special aspect is the stability-oriented demand from automotive manufacturing processes. Especially, the final assembly in automotive manufacturing demands a well-defined process, along with stable in-car software, to minimize variability and ensure consistent results while maintaining assembly efficiency. Therefore, the automotive software for production usage is desired to be stability-oriented. Simultaneously, OEMs are dedicated to speed up the software update frequency to provide car owners with recently developed features, which necessitates flexibility of existing software. Therefore, the design of automotive software faces the challenge of paradoxical requirements due to the flexibility-oriented development and the stability-oriented production processes.

Especially, the µC-based ECUs in the car are going to have long-term usability due to various reasons such as cost discipline and real-time capability (see Section 3.3.3). Furthermore, these ECUs have direct intersection with sensors, actuators, and setpoint generators that are commissioned and tested within the end assembly (see also Section 2.3). Hence, this thesis focuses on software architectural approaches regarding µC-based embedded systems leveraging layered and signal-oriented software architectures.

Since the stakeholders (production and development) of in-car software have paradoxical requirements, the key idea of a production-optimized architecture is to

separate the concerns within software systems and to provide stakeholders with software sub-systems with different focuses.

4.1 Separating the concerns of production and development

4.1.1 Preliminaries

To use a software system within a given ECU with the principle separation of concerns, different *parts* or software sub-systems need to be established, with each addressing a separate concern using a given set of information.

Definition 4.1: Part in automotive embedded software

In this thesis, the term *part* is used to denote the software sub-system within the µC-based automotive embedded software that can contain several Software Components (SW-Cs).

Software *parts* (see Definition 4.1) primarily address the functional aspect of the system architecture within automotive software systems, corresponding the functional Vehicle Systems Architecture (VSA) shown in Figure 2.7. Each software part implements customer features from a specific perspective (block 2.1 in Figure 2.7) and corresponds to several sections of stakeholder requirements (block 1.2 in Figure 2.7).

Additionally, µC-based software systems incorporate middleware, as defined in Definition 3.3. In the context of the AUTOSAR Classic Platform (CP), middleware corresponds to Basic Software (BSW) and Run Time Environment (RTE), as discussed in Section 3.3.1.

Since the CP has become the de facto standard in automotive software development, this thesis adopts AUTOSAR terminology, e.g., RTE and BSW, and visualization conventions to maintain consistency and clarity. However, it should be noted that

the concept introduced in this thesis is not dependent on the implementation of CP. µC-based software systems (Definition 3.1) can be configured and developed without CP or AP, and the proposed concept is equally applicable in contexts that follow model-based development processes (see Definition 3.6).

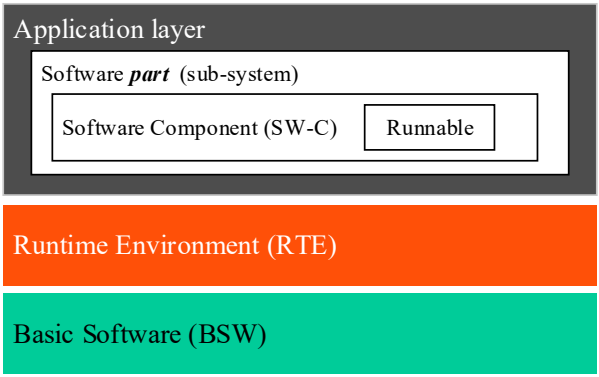


Figure 4.1: Terms and relationships in automotive embedded software systems

A µC-based ECU in the car corresponds to a software system, which can be divided into several *parts*, see Definition 4.1. Interactions and signal exchanges among parts are possible, corresponding to the logical VSA introduced in Section 2.2.2. Furthermore, a software part can contain several SW-Cs that correspond to the dedicated software solution within the physical VSA. Within an SW-C, several Runnables (see also Section 3.3.4) can be executed according to trigger conditions, as Figure 4.1 illustrates.

Against the background, separating the concerns of production and development involves defining and establishing such software parts within the functional VSA of the embedded system that the technical solution of the developed software system within the physical VSA can support decoupled software usage of different stakeholders.

In context of the requirement aspect within VSA, the term *feature* refers to a distinct functionality that interacts with the external environment. These functionalities,

e.g., the Antilock-Braking System, may or may not be experienced directly by the occupant [122]. Regarding the concerns of automotive production and development, the thesis categorizes features in automotive software systems as follows:

- *Customer features:* These are the functionalities that interact directly with occupants and characteristics that an occupant can experience. For example, the functionality that ensures the sunroof opens at a desired speed when the corresponding button is pressed, is a customer feature.
- *Non-customer features:* These are functionalities that do not interact with occupants. Instead, non-customer features interact with manufacturers, workshops, or other expert users, for instance, via a testing device, see also Section 2.4.3. For instance, that the velocity is reported to a testing device when receiving the diagnostic request, corresponds to a non-customer feature.

4.1.2 The basic and customer parts

This thesis proposes handling µC-based automotive embedded systems by focusing on software *parts* rather than the entire software system (see Figure 4.1). Especially, the thesis introduces two terms: *the basic part* and *customer parts*, see Definition 4.2.

In an automotive software system (Definition 3.1), there must be one basic part and there can be several customer parts. The in-line flashing process for vehicular ECUs is executed as a single uninterrupted procedure, leading to the single presence of the basic part. In comparison, features realized by customer parts can be developed by several development teams and are not relevant to procedures within the end assembly. Therefore, the number of customer parts can exceed one to support distributed software development.

Definition 4.2: The basic and customer parts

The **basic part** within the automotive embedded software system is defined as the part for production-related usage, enabling end assembly processes and facilitating the commissioning and testing of production procedures (see Section 2.1.2).

Simultaneously, the **customer part** is defined as a part for production-irrelevant usage, enabling the desired *features* given by the latest requirements, making frequent and regular software updates possible.

Based on the Definition 4.2, necessary components within the basic part are derived as follows:

- *Diagnostic interfaces*: Given that production-related software operations such as commissioning and testing rely on diagnostics (see Section 2.4.3), diagnostic interfaces must be included in the basic part.

  Definition 4.3: Diagnostic interface

  A **diagnostic interface** is defined as a *non-customer feature* of the in-car software that processes specific diagnostic requests, allowing the software to supply information to the connected testing device or to modify data within the in-car software system.

- *Communication interfaces*: The tested and commissioned peripherals (sensors and actuators) can also be referenced by SW-Cs of customer parts. Therefore, inter-part communication interfaces must be provided by the basic part to facilitate the functionalities of customer parts.

The concept of the *basic* and *customer parts* within automotive embedded software systems are designed to address concerns of different stakeholders, particularly the production and the development. In the context of VSA, the following constraints and relationship (shown in Figure 4.2) can be derived:

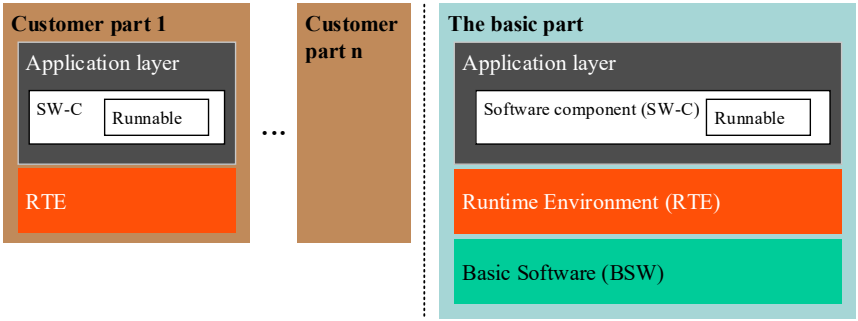


Figure 4.2: Terms and relationships in a software system containing the basic and customer parts

- An ECU corresponds to a software system, which must contain one basic part and can include several customer parts.
- The basic part must contain both the application layer and the middleware (discussed in Section 4.1.3).
- Within the application layer of the basic part, SW-Cs must be implemented to realize *non-customer* and *customer features*, in which
  - The non-customer features enable the usages on the assembly line, such as bootloader and diagnostic interfaces, see also Section 2.4.2, and
  - The customer features guarantee the drivability after the end assembly.
- Customer parts have their own application layers, which contain SW-Cs.
- The SW-Cs within customer parts must implement the customer features that are not covered by the basic part. Additionally, these SW-Cs must realize necessary supporting features required for the customer features within the customer parts.

Additionally, *customer parts* (see Definition 4.2) do not address mandatory customer features and are considered as add-ons for the basic part. One µC-based automotive embedded software system may have no customer parts, such as in the

engine control unit, in which most implemented features are mandatory for vehicle drivability and safety.

From the perspective of automotive production, the basic part corresponds to the software that must be provided during in-line flashing and is preferred to be as compact as possible. In the context of the entire software system, customer parts must implement the requested features that are not covered by the basic part. Additionally, customer part may overwrite the features of the basic part if needed.

4.1.3 Realization of separated concerns

Following the derivation above, the basic and customer parts should be defined and separated within automotive software systems that are µC-based and employ a layered, signal-oriented architecture, as discussed in Section 3.3.3. This separation can be conceptualized and implemented at various levels. Based on VSA shown in Figure 2.7, the following possibilities are identified:

- *Separation at the functional VSA*: Customer and non-customer features are separated, where the implementation in the software system is not considered, as shown in block 2 in Figure 2.7. For instance, dependencies between features can be extracted using algorithmic approaches [168].
- *Separation at the logical VSA*: The allocation of technical solutions is involved, where the code for the basic and customer parts is defined.
- *Separation at the physical VSA*: The technical solutions of the basic and customer parts can be deployed as independent units.

For website development with layered software architecture, separations at physical and logical level are defined regarding web assemblies [155]. For the µC-based software systems, the thesis defines terms of *logical separation* and *physical separation* of software parts as follows:

Definition 4.4: Logical separation

Logical separation stands for software partitioning on the level of the source code. A logically separated software part corresponds to certain SW-Cs, source files, or function groups. The compiler creates one binary file based on all software parts and possibly the middleware (see Definition 3.7). Logical separation aims to simplify code management and readability [155].

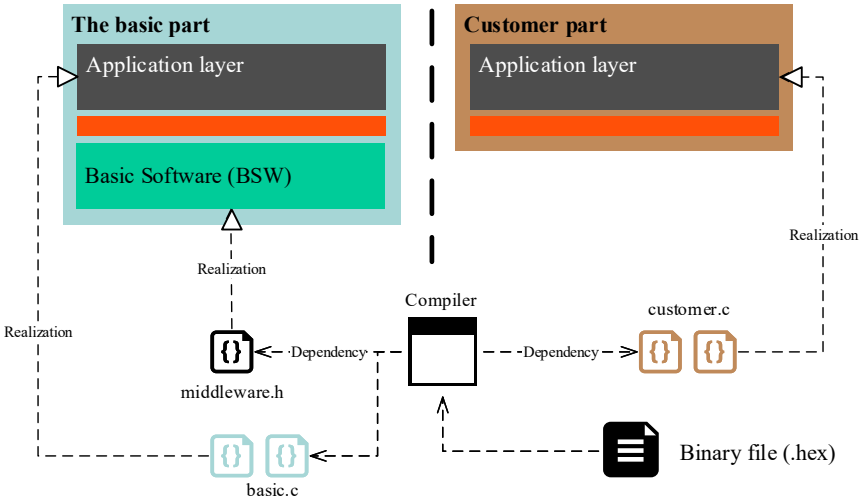
Definition 4.5: Physical separation

Physical separation represents software partitioning on the level of compiled binary files. Each software part is compiled to one binary file that can be loaded onto the µC, where dependencies between the compiled files are allowed. The goal of physical separation is to streamline the development process by isolating the software components during implementation.

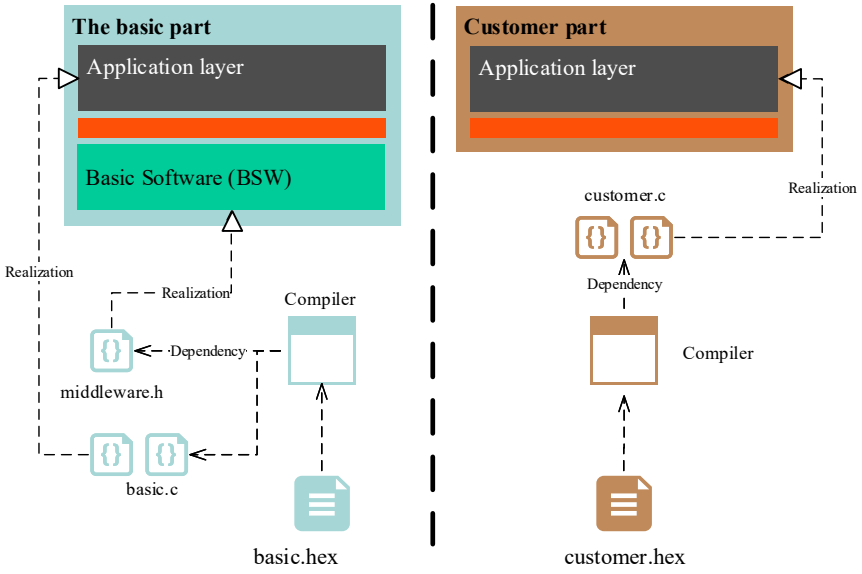
Assuming that the basic part and a customer part should be separated *logically* within an embedded software system, the basic and customer parts contain several SW-Cs respectively. Based on these SW-Cs, header and source files are generated, for instance, *basic.h* and *basic.c* in Figure 4.3-(a). Along with the middleware which belongs to the basic part, the source and header files of the both parts can be compiled by a single compiler to derive an executable binary file, as illustrated in Figure 4.3-(a).

In comparison, physical separation requires a clear boundary within the software system considered, as shown in Figure 4.3-(b). Similar to logical separation, each part contains its respective SW-Cs, with the basic part also including the middleware. However, the basic and customer part must be compiled independently. Additionally, each software part results in its own binary file.

Based on the definitions and short introduction, logical separation is more feasible and less error-prone than physical separation, as it mainly involves source code within embedded systems. The hardware-related challenges of the system, such as memory usage, the memory layout, and the structure of middleware, are not



(a) Logically separated basic and customer parts



(b) Physically separated basic and customer parts

Figure 4.3: Realization of partitioned software: logical and physical separation

necessary to be considered. This kind of refactoring (see Section 3.4) does not change the external behaviors of the system.

The physical separation for a determined embedded system cannot alter the external behaviors regarding the customer features as well. However, carrying out the physical separation involves planning of memory and computing power, instantiation of communication interfaces which are carried out or supported by the compiler in case of the logical separation.

4.2 Production-oriented software partitioning

4.2.1 Concept derivation

As discussed before, the efficiency of automotive production can be ensured by establishing software parts for production within software systems (see Section 4.1), which corresponds to an alteration of the software architecture. Additionally, software parts for automotive production can be separated logically (Definition 4.4) or physically (Definition 4.5).

Against the background, two thoughts arise from the conceptional idea of separation of concerns:

Q1 Can the interest of automotive production regarding software usage be addressed without altering the software architecture?

Q2 Can automotive software systems be separated logically into two parts to meet requirements of automotive production?

Consideration of Q1 Establishing software parts within the existing software systems often involves a change of the architecture of software systems. However, architecture changes, particularly by introducing architectural discipline, can increase the overall complexity of the software system considered, changing the development procedure and raise maintenance costs. Additionally, shifting existing

software systems to a desired architectural style, namely software refactoring, is challenging for automotive embedded systems, as discussed in Section 3.4.

Addressing the raised question Q1, ensuring the interest of production is to maintain and reduce the in-line flashing duration, and to guarantee the results of commissioning and testing during the end assembly, see also Section 2.1.2.

To reduce in-line flashing duration by minimizing the software size (i.e., size of output binary files, measured in Byte) for in-line flashing can be achieved without altering the software architecture. For instance, the measures *production-specific software versioning* (Section 3.1.1) and *shadow memory* (discussed in Section 3.1.3) allow software size to remain constant during regular updates. Additionally, the measure *delta flashing* (discussed in Section 3.1.2) enables flashing only the difference between the current and loaded software versions, which can potentially reduce the software size for in-line flashing by up to 95% compared to a full software update. However, such measures are either associated with high costs or subject to operational conditions. Specifically, usage of shadow memory requires nearly double flash memory, which leads to additional cost of µC-based systems. Similarly, delta flashing may increase flashing duration when difference between the considered two software versions exceed a threshold [172].

Ensuring the stability of the results of commissioning and testing during the end assembly can also be fulfilled by the measures introduced above. The measures *production-specific software versioning* and *delta flashing* can help maintain consistency in automotive ECU software by using a determined software version during the assembly process. However, such stability is limited to short periods or specific ECUs. Production or safety-critical issues may necessitate software updates that override the determined version for the end assembly.

Therefore, current measures for addressing software challenges during the end-assembly process are only partially effective if the in-car software architecture is not optimized for production-oriented use. Without altering the software architecture, the requirements of automotive production regarding software utilization cannot be adequately fulfilled.

Consideration of Q2 Since software partitioning is intended to address the needs of the automotive production in case of frequent and regular software updates (discussed in Section 4.1), logical separation (see Definition 4.4) appears more feasible. Approaches with usage of logical separation restrict the discussion within the logical view of the considered software system, as shown in Figure 3.4, avoiding additional considerations regarding compiler and compilation processes (see also Definition 3.7).

However, the usability of logical separation onto automotive software systems is limited due to the characteristic of µC-based ECUs, where the executable format (e.g., .hex files) does not align with the SW-Cs in the logical view, as illustrated in Figure 3.4. Logical separation only allows for separation of concerns within the logical view. Customer features and non-customer features cannot be clearly distinguished due to the unified .hex file, as shown in Figure 4.3-(a). Additionally, logical separation cannot reduce the software size for the in-line flashing process, as only one binary file is generated.

Regarding the layer-based automotive software systems, e.g., CP discussed in Section 3.3, the dependencies between SW-Cs and the middleware in existing software systems prevent direct assignment of SW-Cs to production-related or irrelevant parts. As a result, logical separation cannot protect the commissioning and testing procedures from regular and frequent software updates.

Although the realization of physically separated embedded software systems demands extra costs, the costs can be rewarded by software usage of binary files, especially regarding the following aspects:

- *In-line flashing*: Assuming that the binary file of the basic part can work independently from those of customer parts, physical separation allows for installation of the basic part during end assembly (see Section 2.1.2). The predictability of in-line flashing can be improved, as the basic part is updated less frequently than the original software system.
- *Software verification*: The commissioning and testing procedures often involve peripherals, as shown in Figure 2.13 and Figure 2.18. Physical

separation allows for the verification of only specific software parts, such as the basic part, decoupling software verification from updates to other parts.

- *Homologation*: Besides the obtainment of a vehicle type approval for vehicular systems and components, OEMs must demonstrate systematic software update realization towards the approval authority to receive a certificate [65]. The homologation process for software updates can be simplified if the binary file of the basic part software remains unchanged during updates.
- *Independent maintenance and development*: Physical separation requires defining inter-part communication interfaces, which incurs additional memory and computing power costs, see Definition 4.5. However, it allows for the independent development and maintenance of software parts, as long as the inter-part communication interfaces remain unchanged.

Concept definition Based on the analysis above, this thesis mainly follows the approach of *physical separation* of automotive embedded software, defining the main concept of the thesis as *Production-Oriented Software Partitioning (POSP)*.

Definition 4.6: Production-oriented software partitioning

The term **Production-Oriented Software Partitioning (POSP)** is defined as the software architectural style for automotive embedded systems, in which one *basic part* and several *customer parts* (Definition 4.2) are separated *physically* (Definition 4.5).

The tendency towards *physical separation* of the proposed concept is driven by software usage within the end assembly of automotive production and the characteristics of µC-based embedded systems.

Comparison with alternative measures Based on the derivation above, the proposed POSP (Definition 4.6) leverages the measure of software partitioning,

especially physical separation (Definition 4.5), to enhance the efficiency of software usage within the end assembly by the following aspects:

- Reduced in-line flashing duration by flashing only the basic part (Definition 4.2), where the size of the basic part is smaller than the original software;
- Enhanced commissioning and testing results by using software features of the basic part, where the basic part is updated less frequently than the original software.

Table 4.1: Comparison of measures aiming to enhance assembly stability regarding software usage

Criterion	**Prod.-oriented sw versioning**	**Delta flashing**	**Shadow memory**	**POSP**
Change on architecture	No	No	No	Yes
In-line flashing duration	Stabilized	Reduced if applicable	Reduced	Reduced
Commissioning and testing	Enhanced in short term	No improvement	Enhanced in short term	Enhanced
Cost	Software management	None	Doubled memory	Physical separation

Compared with the alternative measures (shown in Table 4.1), the characteristics of the proposed concept are highlighted, where:

- *Production-oriented software versioning* (Section 3.1.1) stabilizes in-line flashing duration and commissioning/testing results but provides only short-term benefits.

- *Delta flashing* (Section 3.1.2) reduces the flashing size, but it does not enhance the stability of commissioning and testing when frequent software updates are required.
- *Shadow memory* (Section 3.1.3) offers advantages in both in-line flashing and commissioning/testing but incurs higher production costs due to the additional flash memory.

Thus, POSP trades software development costs for production stability and efficiency. This trade-off is beneficial when production volume is a decisive factor in the value-added chain, such as in the automotive industry.

4.2.2 Design principles

To ensure the feasibility of POSP, software parts within the embedded software system must adhere to the following rules:

1. *Compilability*: Each part in the software system must be compilable, adhering to declared physical separation in Definition 4.6.
2. *Uniqueness and independence of the basic part*: The basic part must be unique within the software system and capable of running the embedded device independently. Specifically, it must provide the functionalities necessary for software usage during end assembly, including in-line flashing, commissioning, and testing (see Section 2.1.2).
3. *Middleware inclusion*: Due to the independence requirement, the basic part must contain the whole or a part of the middleware, as commissioning and testing procedures involve operations of peripherals. Moreover, the basic part must contain SW-Cs that enable software usage within the end assembly and end-of-line tests.
4. *Multiple customer parts*: The embedded software system can include multiple customer parts. Inter-part communication between the basic and

customer parts must be regulated by the basic part to avoid unpredictable behavior in the absence of some customer parts.

5. *Communication interfaces*: The basic part must provide communication interfaces for customer parts and be capable of handling these interfaces in the following scenarios:

   a) When some customer parts are not flashed or included in the software system;

   b) When signals are manipulated by multiple SW-Cs from different software parts.

6. *Customer parts compatibility*: Customer parts must be compilable. However, they do not need to be operational independently and can utilize the communication interfaces provided by the basic part.

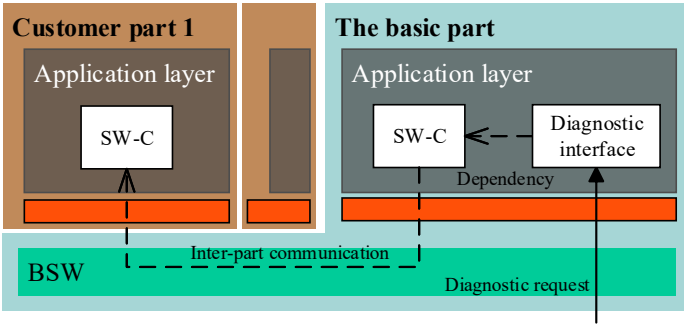
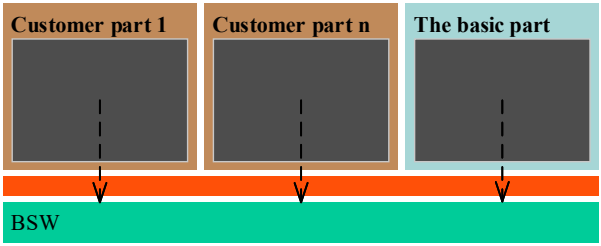


Figure 4.4: Distribution of the basic and customer parts within µC-base automotive ECUs

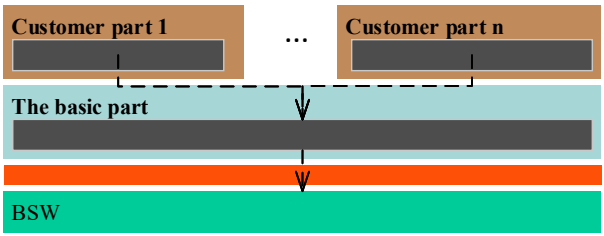
Based on the derived characteristics regarding POSP, a feasible distribution of the basic and customer parts is a reversed L-form, as shown in Figure 4.4. In this distribution:

- The basic part contains the entire middleware, on which the SW-Cs in both the basic and customer parts depend.

- The SW-Cs within the basic part enable the diagnostic interfaces (Definition 4.3).
- Applications within the customer parts require the inter-part communication interfaces provided by the basic part.
- Specifically, in the context of the CP, the realization of partitioned software demands that the customer part encompasses RTE, as discussed in Section A.6.2.



(a) Invalid distribution: the basic and customer parts parallel on the middleware



(b) Invalid distribution: customer parts upon the basic part

Figure 4.5: Invalid distributions of the basic part and customer parts

The following straightforward distributions are not feasible:

- *Parallel distribution*: The basic part and customer parts are built independently upon the middleware, as shown in Figure 4.5-(a). This distribution is not realizable, as the basic part without middleware is not operational in µC-based systems. Furthermore, simultaneous access to a middleware

component by multiple parts requires regulation by the middleware, which necessitates output regulators or orchestration. If the middleware lacks information about the software components of the parts, orchestration and prioritization become challenging.

- *Vertical distribution*: Customer parts are built upon the basic part, as illustrated in Figure 4.5-(b). Although this distribution is feasible regarding compilation (since the basic part with middleware can be compiled as a whole), applications in customer parts do not use the applications in the basic part. Instead, applications in customer parts handle signals of the same category as applications in the basic part. Therefore, customer parts should not be positioned in a higher position over the basic part in a layered software architecture (see Section 3.2.1).

4.3 Design of partitioned software

Within the part distribution derived for POSP shown in Figure 4.4, several issues must be addressed regarding architectural design at the SW-C level. These issues, which are not challenges within SW-Cs in monolithic software design, require consideration in a partitioned context.

Furthermore, the middleware and µC hardware must support and implement the architectural design at the µC level. This section focuses on addressing issues specific to the SW-C level, as the middleware is dependent of the used µC hardware platform.

4.3.1 Inter-part communication

As stated in Section 4.2, inter-part communication must be provided by the basic part and be consumed by customer parts. While it is technically feasible for SW-Cs in customer parts to interact directly with the middleware, this thesis strongly advises against such direct calls. Instead, inter-part communication must always

be mediated through interactions between SW-Cs in the basic and customer parts due to the following reasons:

- Allowing direct calls from customer part SW-Cs to the middleware would necessitate implementing interfaces within customer parts regarding middleware. This introduces additional costs.
- If both the basic and customer parts require access to the same middleware module, the decision-making regarding priority would have to be handled within the middleware. However, this is neither technically feasible nor aligns with the middleware's intended role.

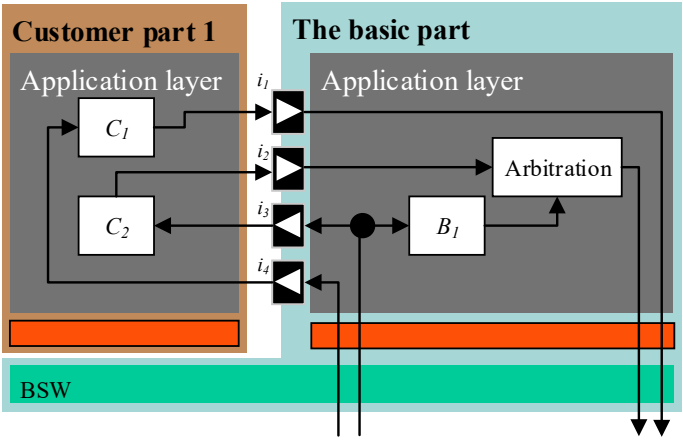


Figure 4.6: Inter-part communication within POSP

Assuming SW-Cs upon the middleware handle signals exclusively in the form of input and output, meaning that the communication between SW-Cs is realized by Sender or Receiver (S/R) mechanisms, types of inter-part communication interfaces can be categorized based on signal usages.

Given that signals have only two possible directions, input or output, and can be used by either a single part or multiple parts, there are four types of inter-part communication interfaces:

- *Forwarded input:* A signal received by the basic part and passed along to customer parts. As the middleware is allocated within the basic part, any signal used by customer parts must be forwarded by the basic part. For instance, a voice-controlled request for the ambient light is not necessary for the basic part must be forwarded to the dedicated customer part, as illustrated by interface i_4 in Figure 4.6.

- *Forwarded output*: A signal output generated by a customer part and forwarded to the middleware by the basic part. For example, the result of voice-controlled ambient light request should be reported to the upper machine, e.g., the body controller. This response is irrelevant to the basic part and should be forwarded to the corresponding communication channel through the middleware, as i_1 in Figure 4.6 illustrates.

- *Shared input:* A signal used by the basic and customer parts. An example is the measured rotating velocity of the vehicular power window motor that is necessary for both production usage and customer applications. Such signals are used by several SW-Cs within the basic and customer parts, like i_3 in Figure 4.6.

- *Conflicting output:* Signals from SW-Cs of different parts have the same type and aiming at the same block in the middleware. For instance, the basic part needs to control the motor of vehicular power window, to text its functionality. However, a customer part also requires control over this motor, which leads to a conflict that multiple SW-Cs access the same output channel with different desired values. In this case, the signal from customer parts via an inter-part communication interface is referred to as a conflicting output, as illustrated by i_2 in Figure 4.6. Conflicting outputs must be handled by an arbitrator (Definition 4.7) that determines the final output sent to the middleware.

In the following context, the interface symbol (e.g., i_1 in Figure 4.6) represents one or more S/R signals, each characterized by its format, value, and cycle time.

Given the signals of inter-part communication only have two directions (input and output), and two usage form (used by single part or multiple parts), the above introduced type of inter-part communication interfaces are exhaustive. The signals used only within the basic part are not involved for inter-part communication.

4.3.2 Arbitration mechanisms

As discussed in the last section, partitioned software can cause conflicting access of output blocks of the middleware. For instance, the motor of the vehicular power window system can be accessed by different applications, such as the diagnostic command, the stop request from trap protection module from the basic part, the button input, or the voice-controlled request from a customer part. The conflicting inputs from different sources must be handled to ensure the proper behavior of the system. To resolve conflicting signals, three types of arbitrators (Definition 4.7) can be utilized: *arbitration with a mode variable, arbitration with a state machine,* and *arbitration with input priorities*. These mechanisms enable determination of appropriate output signals to the middleware without knowledge of the SW-Cs that generate the conflicting signals.

Definition 4.7: Arbitrator

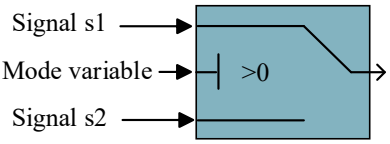
In this thesis, **arbitrator** is defined as the algorithm for resolving *conflicting outputs* from different software parts when using POSP.

Arbitration with a mode variable Mode variables (shown in Figure 4.7-(a)) are commonly used in automotive software systems. For instance, the extended diagnostic session [82] (see also Section 2.4.3) is activated and deactivated by a mode variable, which is further triggered by a secured verification process.

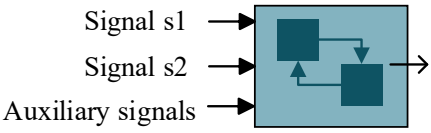
When conflicting signals are managed by a mode variable in POSP, it is proposed to define the mode variable in relation to the software parts. For example, a partitioned software system with a basic part and one customer part can use the following enumeration to trigger the arbitrator:

```
enum modeVariable{
  BasicPartOnly = 0,
  CustomerPartActivated = 1,
  CustomerPartDeactivated = 2
};
```

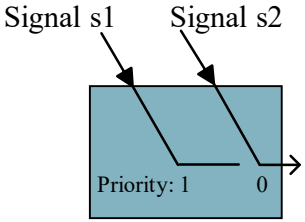
The mode variable can be triggered by system states, such as a variable stored in non-volatile memory indicating that the ECU is not initialized after software re-flashing. Therefore, the mode variable is set to 0 during the initialization. Alternatively, the mode variable can be manipulated by diagnostic requests.



(a) Arbitration with a mode variable



(b) Arbitration with a state machine



(c) Arbitration with input priorities

Figure 4.7: Arbitration mechanisms for conflicting signals

Arbitration with a state machine In case that an output involves several stakeholders, the usage of mode variables can be extended to a finite state machine, whose outputs depend not only on the current inputs, but also the current state of the machine [123], as shown in Figure 4.7-(b). Using state machines allows for flexible output management by considering additional signals within the software system. In addition to the global mode variable mentioned earlier, local mode variables tied to a specific customer feature, system error state, or the operational

status of actuators can be used to trigger the state machine. State machines also enable time-dependent output switching, allowing the basic part to control the output for a certain duration before an application of a customer part takes over.

Arbitration with input priorities Since the arbitrator does not have detailed knowledge of the connected SW-Cs, one approach to resolve output conflicts is to extract information from the SW-Cs. Signals or connected SW-Cs can be declared with priorities, which the arbitrator uses to determine the output. This approach is similar to the interrupt vector table [96] used in µCs, where different interrupt services are called according to their predefined priorities.

In the case of conflicting outputs, signals from applications of the basic part can be given higher priority, as depicted in Figure 4.7-(c). This method ensures that essential signals from the basic part are prioritized over those from customer parts.

Discussion The mechanism of *arbitration with mode variables* is considered within the following context due to ease of implementation:

- Arbitration with mode variables only requires a global variable and SW-Cs for the corresponding conflict signals. In the case of a series software system, SW-Cs for arbitration can be extended to state machines to handle various modes.
- When implementing arbitration with input priorities, additional considerations must be made for the SW-Cs for arbitration. These include factors such as dead time, signal receiving mechanisms, and other related aspects to ensure proper functioning.

4.3.3 Realization of diagnostic interfaces

In a monolithic automotive software system, applications built upon the middleware must consider the interests of development, customers, and production. The

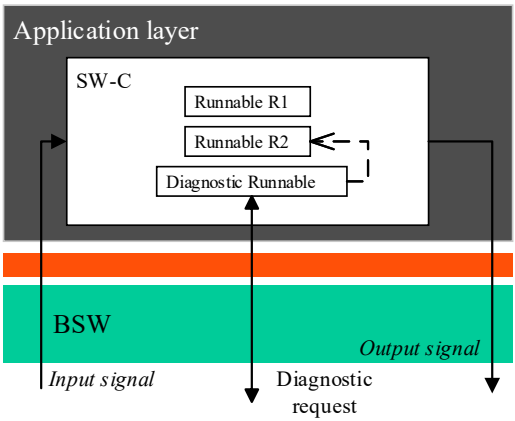
realization of diagnostic interfaces (see Definition 4.3) in such a system relies on the SW-Cs for customer features.

This development methodology results in a subordinate relationship between the application SW-C and the diagnostic interface realization. Specifically, diagnostic interfaces are realized by Runnables (introduced in Section 3.3.4) within the application SW-C, as depicted in Figure 4.8-(a). Consequently, the application SW-Cs depend on input/output SW-Cs and the middleware. Additionally, the diagnostic-relevant realizations within application SW-Cs further depend on services within the middleware. This subordinate relationship is one reason why diagnostic interfaces are fragile: updates to algorithms within the application can cause issues in the final assembly, as illustrated in Figure 3.16.

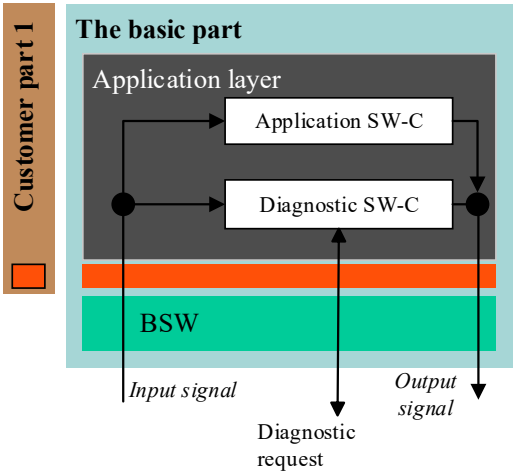
The concept of POSP (Definition 4.6) aims to decouple production and development concerns, with the basic part designed for production usage. Thus, the realization of diagnostic interfaces in the basic part can be prioritized over application SW-C during architectural design.

Furthermore, the thesis recommends implementing diagnostic interfaces solely within the basic part, excluding customer parts from assuming responsibilities for off-board diagnostics. The reasons for this exclusion are summarized as follows:

- As the basic part realizes all features related to the end assembly and safety-relevant issues. Therefore, the diagnostic interfaces provided by the basic part is sufficient for addressing hardware-related issues, which is essential for the repair and maintenance tasks in after-sales.
- Similar to the considerations regarding inter-part communication, see Section 4.3.1, implementing diagnostic interfaces within customer parts introduces unnecessary complexity.
- For extended function diagnostics, the development teams must be equipped with additional tools for functional verification, such as JTAG interfaces [119].



(a) Diagnostic modules in form of Runnables within the SW-C



(b) The proposed distribution of diagnosis-relevant function: diagnostic modules in form of SW-C

Figure 4.8: Allocation of diagnostic relevant functionalities for the basic part software

- Even if critical information is located within customer parts, such signals can be forwarded to the core part (e.g., signal i_4 in Figure 4.6) and accessed via diagnostic requests.

Therefore, it is proposed to separate the diagnostic-relevant functionalities from application SW-Cs and to implement diagnostic-relevant functions in form of SW-C, as shown in Figure 4.8-(b).

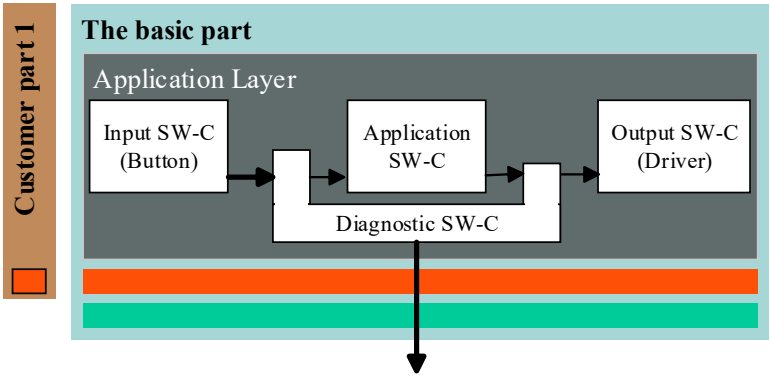


Figure 4.9: Example realization of a diagnostic SW-C in the basic part software

Separating the responsibilities of application and diagnosis provides a framework for shaping the realization of diagnostic interfaces. In this context, the thesis proposes to allocate the diagnostic SW-C around the application SW-C that need to be commissioned or tested, as illustrated in Figure 4.9. By applying this style, the application-relevant operations can be decoupled from the middleware by the diagnostic SW-C.

From the perspective of the end assembly, especially in embedded systems, the primary concern is often whether the system works, rather than how well it works, as functioning peripherals indicate proper installation. For instance, adjusting and optimizing the vehicular ambient light requires careful considerations and precise algorithms [30], although production typically focuses only on whether these LEDs function correctly.

More precisely, consider an SW-C that receives input signals from input SW-Cs, such as SW-C *Button* in Figure 4.9 that processes these signals using algorithms, and generates outputs towards other SW-Cs, such as *Driver*. The diagnostic SW-C surrounding the application SW-C should be capable of manipulating these inputs and outputs if necessary. If the algorithm within the application SW-C is essential for the functionalities of other SW-Cs of the ECU, the diagnostic SW-C can forward the inputs from the input SW-Cs and the outputs to the output SW-Cs.

As the diagnostic SW-C does not organizationally belong to the application SW-C, the formulation of diagnostic requests can be more straightforward. For instance, data acquisition during the calibration process in the end assembly (e.g., transmitting sensor values) can be realized without using the application SW-C, as shown by the signal flow shown by bold arrows in Figure 4.9.

The application-surrounding design of diagnostic interface realization isolates the application SW-C and supports SW-Cs from the middleware. This isolation allows for black-box testing of the application SW-C as well as testing of input/output peripherals without referencing the application SW-C. However, this design demands more signals within the software systems and more memory usage. For flexible usage of the diagnostic interfaces, logic and algorithms need to be built within the diagnostic SW-Cs.

To conclude the realization of diagnostic interface, POSP enables and encourages the separation of diagnostic-relevant and application-relevant functionalities at the SW-C level. This separation allows for a flexible formulation of diagnostic interfaces, which can be addressed in advance of software architectural design.

4.3.4 Resource usage

Depending on realization mechanisms of software partitioning and the utilized µC, the allocation of the basic part and customer parts after successful compilation can vary with respect to the hardware, see also Section 2.4.2. In principle, the usage

of µC resource should be taken into account in advance of allocation of software parts in the software system:

- *Memory usage*: In principle, the memory usage of each software part should be continuous and firmly defined before compilation. Continuity is required so that the memory tests (such as CRC tests carried out in Atmel µC [12]) can be carried out. Additionally, the definition of upper and lower memory boundaries is demanded to avoid stack overflow of one software part affecting another.

  Given that the basic part contains the middleware, and assuming the first released version for the basic part corresponds to a binary file of x kB, we propose that at least $0.2x$ kB should be reserved for the further development of the basic part. Additionally, $0.5x$ kB should be reserved for allocation of customer parts[1].

- *Computing power usage*: As the basic part contains the middleware, the scheduling of periodic function calls is allocated within the basic part, limiting the computing power available to customer parts. Therefore, an analysis of computing power consumption for possible periodic functions is necessary to ensure that real-time requirements for these functions can be met in customer parts.

Due to the diverse configurations of compilers and µC architectures, the topic of resource usage is further explored in Chapter 6 with a focus on the selected software platform and µC. Particularly, additional resource costs are caused when implementing new architectural disciplines, such as software partitioning. These costs are highly influenced by the compilation and linking processes.

[1] The proposed allocation scheme is a design heuristic. It is informed by analysis of memory consumption in some series ECU and aligns with the common practice of allocating headroom further development.

4.4 Discussion

4.4.1 Theoretical analysis of the introduced concept

Based on the analysis in Section 4.3 and proposed usage, POSP can influence the development process of automotive software in the following aspects:

- *Flexible and independent application design*: By implementing the required customer features within customer parts, these functions can be developed independently of production usage, as long as the inter-part communication interfaces remain unchanged. Additionally, the realization of customer features can be distributed across different customer parts, each of which can be compiled and tested independently with the support of the basic part. This promotes modularity and ease of maintenance.
- *Fast and stable software update*: Software updates for purchased vehicles require stability and efficiency to minimize downtime and reduce data transmission costs between OEM data servers and vehicles. As long as the basic part remains stable, software updates for specific ECUs can be simplified to updating only the relevant customer parts, thereby reducing the amount of data transferred and verified within the ECU.
- *Accelerated software development*: The independent compilability of software parts provided by POSP enables small-scale development and verification, reducing the time required for code generation and compilation. For instance, generation of the Run Time Environment (RTE) for some large ECUs (such as the vehicular body controller) can take several hours. Partitioning embedded software systems into several compilable parts can shorten the development time, especially when only minor changes are needed.

However, implementing partitioned software also introduces certain challenges and costs in the development process. These challenges should be considered before

refactoring an existing software system or developing an ECU with approaches of POSP:

- *Resource usage*: Adopting POSP consumes additional resources. Additional signals and SW-Cs may be necessary to ensure the functionality of the corresponding software parts. Moreover, resource usage needs to be planned in advance to ensure the feasibility of functionalities in customer parts, see also Section 4.3.4.
- *Stabilizing the inter-part communication*: The independent development and implementation of the basic and customer parts necessitate stable inter-part communication interfaces. These inter-part communication interfaces must be designed with future use in mind to prevent the need for major revisions due to minor updates. However, it is challenging to predict which signals should be included in the inter-part communication interfaces during the early stages of development, which can complicate interface design.
- *Management of software*: Managing software parts instead of a software system increases the cost of software maintenance. The increased number of software entities in a given ECU adds complexity to software version management and product line management. In case that a software system or part is to be updated, one has to make sure that new version interoperate with the old version [37]. The vast configuration space created by the options provided by OEMs (more than 10^{100} possible valid configurations for a product line [84]) can make variability management for the basic and customer parts more challenging than for a monolithic software system.

4.4.2 Usage scenarios of partitioned software

As introduced in Section 4.2, physically separated software allows for partial flashing in the end assembly line.

When embedded software is developed and compiled as a monolithic binary file, the life cycle is restricted, as the software can only be managed as a whole.

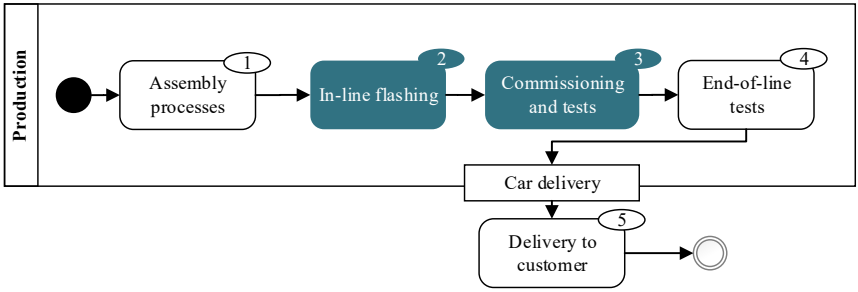


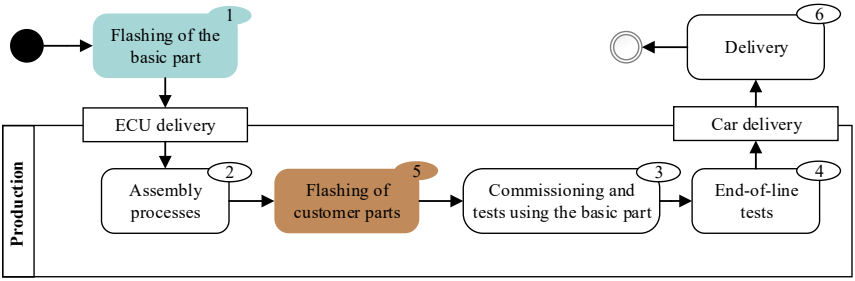
Figure 4.10: The conventional life cycle of the automotive embedded software, cf. Figure 2.3.

Conventionally, the supplier produces ECUs, initializes the memory, and flashes the bootloader onto the ECUs, as illustrated in Figure 4.10 and discussed in Section 2.1.2.

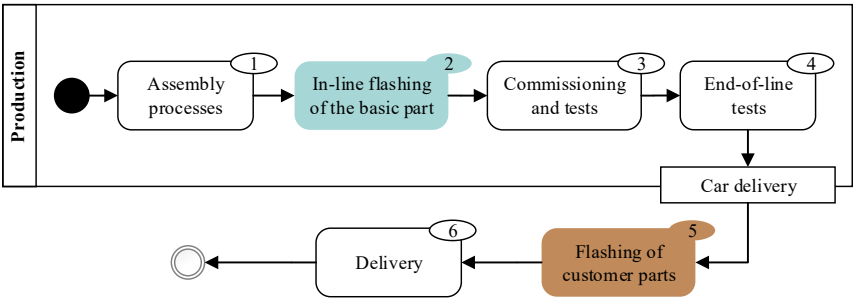
This thesis suggests two usage scenarios that leverage the partitioned software systems:

- *Pre-flashing of the basic part*: In this scenario, the basic part is installed at the supplier before being delivered to the OEM, shown in Figure 4.11-(a). Customer parts are installed on the assembly line to facilitate newly developed features.
- *Postponed flashing of customer parts*: The basic part is installed on the assembly line, facilitating the commissioning process, illustrated in Figure 4.11-(b). Customer parts are installed on to ECUs after end-of-line tests, ensuring flexibility and late-stage customization.

Assuming that the arbitrators (see Figure 4.6) activate the basic part during the final assembly stage, the commissioning and testing procedures for both scenarios can be conducted independently of SW-Cs of customer parts. This independence ensures separation between diagnostic handling and customer features. As long as the basic part remains unaltered, the diagnostic interfaces are decoupled from the applications in the customer parts. Consequently, the concept of POSP theoretically



(a) Usage scenario 1: Pre-flashing of the basic part



(b) Usage scenario 2: Postponed flashing of customer parts

Figure 4.11: Application scenarios of partitioned software systems according to POSP

addresses the challenge of application-dependent diagnostic interfaces (see also Section 3.5.1).

Regarding the topic of in-line flashing, these introduced usage scenarios aim to reduce the sizes of software systems that are installed within the end assembly. The principle behind reducing in-line flashing time relies on the following assumptions:

(A_1) The in-line flashing duration is *linearly dependent* from the installed software size.

(A_2) The size of the basic part is smaller than the size of original monolithic system.

(A_3) The aggregated size of all customer parts is smaller than the size of the original monolithic system.

In practice, assumption A_1 holds if the software size for the target ECU is sufficiently large. Internal data indicates that approximately 65%[2] of the total in-line flashing time is spent on data transfer, which scales linearly with software size. Other in-line flashing tasks, such as memory erasure and verification, are also influenced by the size of data transferred. Only tasks like ECU identification, establishing a diagnostic session, and changing authentication states, are independent of software size. Therefore, this thesis adopts A_1 as a valid assumption.

The validity of assumption A_2 depends on the implementation of POSP. Although the basic part may include a larger middleware compared to the monolithic system, A_2 holds if the application layer of the derived basic part is significantly smaller than that of the monolithic software.

Assumption A_3 is generally valid, as customer parts typically exclude middleware and contain fewer features than the monolithic system.

4.4.3 Expert insights

During the work of this doctor thesis, the concept *production-oriented software architecture* was presented to industrial practitioners. A survey in the form of a set of personal interviews was conducted, involving engineers from diverse sectors, as shown in Table 4.2. Questions used in these interviews are listed in Table 4.3. The interviewees represent the following roles: enterprise software architects, software update engineers, software developers for software platforms and ECUs, electronic engineers for the end assembly, and after-sales specialists.

The feedback from these industrial practitioners is twofold, highlighting the challenges of introducing a production-optimized automotive software architecture and

[2] The figure is derived from a quantitative time analysis of in-line flashing cycles in 2024 from a specific car series.

Table 4.2: Excerpt of interviewees - roles and years of experience

Role	Experience in years
Software developer infotainment module	> 8
Software architect	> 8
After-sales manager	> 8
Software developer update strategy	$6 - 8$
Software developer communication module	$3 - 5$
Software architect	$3 - 5$
Software requirement engineer	$0 - 2$

emphasizing the importance of considering manufacturing processes throughout the software life cycle. Specifically:

- Approximately half of the interviewees from software development sectors expressed skepticism regarding the added value of a production-optimized automotive software architecture. Their concerns include:
  - The design of current software architecture for µC-based automotive ECUs is constrained by stringent requirements and boundary conditions. A fundamental change may not be feasible.
  - Architectural changes often result in increased development costs, which may not be compensated by the potential efficiency gains within the end assembly.
- The remaining interviewees from software development sectors acknowledged the potential of the introduced concept, as they could observe the potential of software partitioning for a distributed software development process. However, these interviewees did not provide feedback regarding the potential improvements in production efficiency and stability.

- The introduced concept is enthusiastically welcomed by staff from the production and after-sales sectors, who suggested numerous use scenarios during the interviews. Nevertheless, these engineers did not provide feedback on the feasibility of implementation or the boundary conditions required for realizing the concept (see Section 4.2 and Section 4.3).

Table 4.3: Question list of expert interviews

Num.	Question:
1	What is your expectation for future automotive software (architecture)?
2	How do you see service-oriented architecture?
3	Is the partitioning of software packages for your job meaningful? What are the challenges to separating production-relevant and customer-relevant parts of automotive software?
4	How often do you observe software updates? What is your opinion towards current software update frequency and strategy?
5	What is the main driver for the complexity of E/E architectures?
6	Can you give three keywords for future-oriented automotive software architecture?

The results of the survey conducted reveal the gap between the stakeholders within the life cycle of automotive software, where:

- Software development teams often neglect the efficiency of software usage during production and after-sales. Most software developers have never even visited one production line, which limits their understanding of production needs.
- Conversely, production and after-sales sectors are keen to improve in-car software to address their specific requirements. However, they lack technical expertise to provide feasible statements, particularly concerning software architecture.

No individual or team can be expert in all disciplines related to automotive software. Moreover, gaining a comprehensive understanding of the realization and mechanisms of automotive software is challenging, for instance, the AUTOSAR Classic Platform (CP) [11]. Therefore, the decision to adopt a production-optimized software architecture for automotive ECUs should be made by leaders capable of influencing both software development and production sectors.

Additionally, feedback from industry practitioners reveals several challenges in implementing the proposed production-oriented software architecture, particularly from the following perspectives:

- How can a software system be evaluated for its suitability to be refactored towards POSP?
- If a system is refactored according to the proposed distribution, what metrics can be used to assess the effectiveness of the refactoring?
- Assuming such an evaluation is possible, what benchmarks or criteria can be used to determine whether the refactoring reaches a break-even point?

Therefore, methodologies must be developed for the concept POSP. Furthermore, these methodologies must provide practical workflows, evaluation metrics, and benchmarks to assess the cost-effectiveness, and added value of refactoring efforts in real-world industrial contexts.

5 Methodology for Production-Oriented Software Partitioning

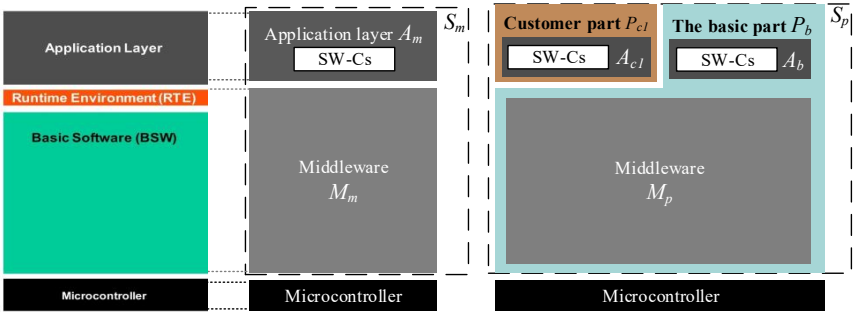
In Chapter 4, the concept *Production-Oriented Software Partitioning (POSP)* is introduced, see Definition 4.6 and Figure 4.4. The proposed concept outlines a desired architecture for µC-based software systems (see Definition 3.1).

However, existing software systems for automotive ECUs are well-established, with long development histories. Developing software systems adopting POSP from scratch is time-consuming and costly, as newly developed mechatronic systems must undergo seamless verification and validation, as discussed in Section 2.2.1. Given this context, there is a need for methodologies that adapt existing legacy software systems to fit POSP.

In the following context, the considered software systems for automotive ECUs are assumed to follow a layered, signal-oriented architecture (see Section 3.3.3).

In the current legacy software systems, all SW-Cs, along with the middleware, are compiled into a single executable file (Definition 3.7). Furthermore, there is no separation within the application layer. These systems are referred to as monolithic software systems, denoted as S_m, where S stands for software system and m for monolithic (see Figure 5.1-(a)).

In contrast, POSP proposes a structure where the basic part contains the middleware, and customer parts rely on inter-part communication interfaces provided by the basic part. A valid distribution takes the form of an inverted L, as depicted in



(a) Comparison between the monolithic software system (denoted as S_m) and the layered software architecture of CP, based on AUTOSAR [22].

(b) The desired partitioned software system S_p, comprising the basic part P_b and a customer part P_{c1}

Figure 5.1: Initial and desired state for the methodology for POSP

Figure 5.1-(b)[1] and discussed in Section 4.3. Software systems adopting this suggested architecture and distribution are referred to as partitioned software systems, denoted as S_p (with p indicating partitioned). Furthermore, the basic part software system within S_p is denoted P_b, where P stands for Part (Definition 4.1) and b for the Basic part. The i-th customer part within S_p is denoted as S_{ci}, with c representing Customer parts.

The transformation from a monolithic software system to a partitioned software system containing the basic and customer parts must be carried out without affecting the external behavior, indicating a refactoring process for the considered software system (Definition 3.8). Although software refactoring techniques and tools are extensively discussed for object-oriented software in various literatures [1, 29], these techniques are not readily applicable to automotive software development. One reason is the use of model-based development (see Section 3.3.4), which is associated with high redesign costs [137]. Additionally, the use of procedural languages, particularly C, poses challenges in a legacy environment [51].

[1] To simplify the following analysis, the RTE of the CP is implicitly categorized as part of the middleware and will not be addressed separately.

5.1 A generic simplified software system

The design of SW-Cs for automotive ECUs can become highly complex, exampled by large Simulink models containing over 15,000 building blocks [149]. To simplify the discussion and the derivation of the methodology, the following assumptions are made:

1. There are three types of SW-Cs in the considered monolithic software systems: *input SW-C* (denoted as W_I in the following context, in which W stands for SW-C), output SW-C (presented as W_O), and *application SW-C* (referred to as W_A), see Figure 5.2, where

   - W_I transforms electrical signals provided by the middleware into explainable signals that can be processed by W_A. For instance, a device's temperature can be measured using a temperature-sensitive resistor, with the temperature determined via a reference table or curve [7]. The corresponding W_I converts the voltage signal into a temperature value and stabilizes it using a low-pass filter, if necessary.
   - W_O transforms explainable signals into electric signals that are used by middleware. For instance, a W_O transforms a period into digital signals to control a blinking LED.
   - W_A performs algorithmic calculation using the provided signal interfaces from input/output SW-Cs. Additionally, diagnostic interfaces are implemented within W_A, see also Section 4.3.3.

2. The communication between these SW-Cs is based on Sender/Receiver interfaces.

3. SW-Cs can have hierarchical levels, meaning a SW-C may consist of other SW-Cs. These SW-Cs are referred to as *CompositionSwComponentType* according to AUTOSAR terminology [18]. In this thesis, the term SW-C is also used to refer to SW-C compositions.

4. SW-Cs can have multiple tasks (known as Runnables in AUTOSAR terminology) that can be triggered by different conditions, see also Figure 4.1.

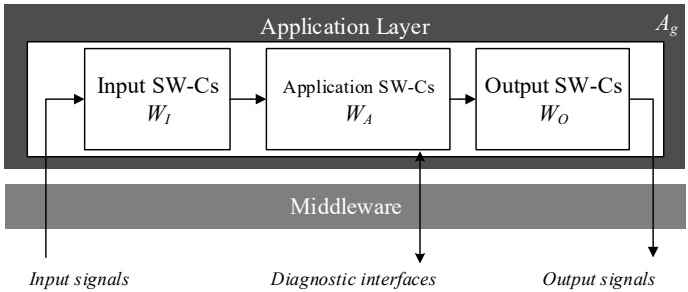


Figure 5.2: A generic simplified model (A_g) of software systems at the application layer for µC-based automotive ECUs

Following the assumptions above, any automotive software system obeying assumptions 1 – 4 can be described by a generic software model A_g (A for application layer, g for generic) in combination with middleware, shown in Figure 5.2. For ECUs covering multiple function aspects, software systems can be represented as a combination of A_g, where all SW-Cs in the system share the same middleware.

The embedded software system of an automotive ECU which is not developed according to POSP, namely monolithically, denoted as S_m, contains a singly configured middleware (referred to as M_m with M representing middleware) and the application layer. The SW-Cs at the application layer consist of multiple software sub-systems A_{mi}, each can be represented by A_g. This relationship is expressed as:

$$S_m = \{M_m, A_{m1}, A_{m2}, ..., A_{mn}\} \tag{5.1}$$

$$A_{mi} \sim A_g \tag{5.2}$$

in which n is the number of software sub-systems at the application layer, m indicates the software system follows a monolithic architecture, as shown in Figure 5.1-(a). The software sub-system S_{mi} have input and output signals, and diagnostic interfaces. The number of these signals and interfaces are represented by

$|I(A_{mi})|$, $|O(A_{mi})|$, and $|Diag(A_{mi})|$, similar to the presentation in Figure 5.2. After compilation, the size of software (denoted by $s()$) measured in Byte is presented as $s(S_m)$. By deleting all SW-Cs within the software system and compiling it again, the size of middleware $s(M_m)$ can be determined.

Based on the presentation of S_m given in Equation 5.1, this thesis suggests two metrics to indicate the production relevance of monolithic software systems:

Definition 5.1: Production Usage Ratio (δ_P)

$$\delta_P(S_m) = \frac{\sum_{i=1}^{n} |Diag(A_{mi})|}{\sum_{i=1}^{n} (|I(A_{mi})| + |O(A_{mi})|)} \tag{5.3}$$

Within Definition 5.1, $|Diag(A_{mi})|$ denotes the number of diagnostic interfaces, $|I(A_{mi})|$ and $|O(A_{mi})|$ represent the number of input and output signals, respectively (see Figure 5.2). A lower δ_P suggests less involvement in commissioning and testing procedures, indicating potential for refactoring.

Definition 5.2: Middleware Ratio δ_M

$$\delta_M(S_m) = \frac{s(M_m)}{s(S_m)} \tag{5.4}$$

In Definition 5.2, $s(M_m)$ is the middleware size, and $s(S_m)$ represents the size of the monolithic software system, measured in kilobytes (kB) or megabytes (MB). A high δ_M indicates that the application layer constitutes a small portion of the software.

Furthermore, a partitioned software system S_p (illustrated in Figure 5.1-(b)) adhering to POSP can be described by software sub-systems like A_g accompanied by corresponding middleware. More formally:

$$
\begin{aligned}
S_p &= \{P_b, P_{c1}, ...P_{ci}, ..., P_{cn}\} \\
P_b &= \{M_b, A_b\} \\
P_{ci} &= \{A_{ci}\}
\end{aligned}
\tag{5.5}
$$

Here, P_b, P_{ci} represent the basic and customer parts, correspondingly. The basic part consists of its middleware M_b and the application layer A_b, each customer part P_{ci} only consists of SW-Cs at its application layer that is denoted as A_{ci}.

5.2 Workflow of refactoring legacy software systems

To refactor existing automotive software systems, represented similarly to S_m (Figure 5.1-(a)), into production-oriented partitioned systems S_p (see Figure 5.1-(b)), this thesis proposes a generic workflow tailored for automotive ECUs that face inefficiencies in software utilization within the end assembly. In the following context, *refactor* is referred to as the transition of POSP (Definition 4.6).

Considering the characteristics of microcontroller-based (µC) software systems and automotive production processes, the proposed workflow integrates a software refactoring approach (see Section 3.4) with principles from automotive production. The workflow comprises the following key steps:

1. Identification of SW-C set to be refactored;
2. Software refactoring with a redundancy-based approach (discussed in Section 5.3);
3. Deployment and verification of the derived software system;
4. Evaluation of internal and external behaviors.

The proposed workflow is outlined in the pseudo code presented in Algorithm 1. A detailed description of the workflow is provided in the following sections.

Algorithm 1 Workflow to refactor software systems towards PoSP

Data: $S_m = \{M_m, A_{m1}, A_{m2}, ..., A_{mn}\}, |I(A_{mi})|, |O(S_{mi})|, |D(S_{mi})|$
Result: $S_p = \{P_b, P_{c1}, P_{c2}, ...P_{cn}\}$
Gets $\delta_P(S_m)$ and $\delta_M(S_m)$
if $\delta_P(S_m) > \delta_{P1}$ **or** $\delta_M(S_m) > \delta_{M1}$ **then**
    Stop(); ▷ Early termination
else
    $M_{p0} \leftarrow M_m$
    for $i = 1$ to n **do**
        repeat
            $(M_{pi}, A_{bi}, A_{ci}) \leftarrow R_r(M_{p(i-1)}, A_{mi})$
            $S_{pi} \leftarrow \{M_{pi}, A_{b1}, A_{c1}, ..., A_{bi}, A_{ci}, A_{m(i+1)}..., A_{mn}\}$
        until externallyEquivalent (S_{pi}, S_m)
    end for
    Gets $r_f(S_{pn}, S_m)$ and $\delta_{pr}(S_{pn}, S_m)$
    Output $S_p \leftarrow S_{pn}$
end if

5.2.1 Identification of SW-C set to be refactored

Premium car models can incorporate over 150 ECUs [165]. Not all of these ECUs are suitable candidates for refactoring into production-oriented partitioned software. For instance, motor ECUs (also referred to as power ECUs [89]) predominantly implement production-relevant features due to drivability in end-of-line testing, making them suboptimal targets for refactoring. This raises the following question: How to identify the potential of a determined ECU regarding POSP?

The potential of a determined ECU for POSP can be assessed based on its relevance to production usage. To illustrate this, two extreme examples are considered:

- SW-Cs within the ECU c_1 are fully production-relevant (Definition 2.1), meaning functions of each SW-C within c_1 must be commissioned and tested during the end assembly.

  - Such ECUs are not suitable for POSP because customer parts to be derived would be empty. As a result, there is no practical benefit in partitioning the software for these ECUs.

- ECU c_2 is production-irrelevant, meaning no diagnostic interfaces are required within the software system. Consequently, the basic software part to be derived does not need to include any SW-Cs at the application layer. The production process only involves flashing the middleware, indicating potential for partitioning.

  - However, there is no need to partition such software systems, as the entire system can be flashed after the end assembly.

  - Once c_2 gains production relevance, its production-irrelevant features can be located into customer parts. Therefore, an ECU with lower production relevance has more potential regarding usage of POSP.

Nevertheless, determining the production relevance (Definition 2.1) of SW-C sets is often non-trivial. Instead of accessing the production relevance of a determined Software Component (SW-C), the thesis suggests evaluating the type and amounts of signals used in the considered software system to estimate the potential of software refactoring towards POSP.

The workflow starts with identification of relevant ECUs (Line 1 of Algorithm 1). When evaluating an ECU with the software system S_m, the potential of POSP must be assessed. It is suggested to examine the production usage ratio δ_P and middleware ratio δ_M.

If either δ_P or δ_M exceeds its respective threshold, S_m is seen as unsuitable for POSP, as stated in Line 2-3 of Algorithm 1. To support the examination of early termination of refactoring legacy software systems, this thesis proposes the following reference thresholds for microcontroller-based systems, considering the use of AUTOSAR Classic:

$$\begin{aligned} \delta_{Pth} &= 0.3 \\ \delta_{Mth} &= 0.7 \end{aligned} \tag{5.6}$$

These presented reference values are intended as reference within this work and should be adjusted based on the specific use case and the target platform.

A high value of δ_P indicates that the majority of required features of S_m must be commissioned during production, meaning fewer SW-Cs can be outsourced from S_m to customer parts P_{ci}. Moreover, a high value of δ_M implies a small application layer compared to the middleware. Since customer parts do not include the middleware, a large middleware share results in a smaller derived share for the customer parts P_{ci}, leading to a poor production reduction ratio δ_{pr} in the refactoring process.

While the metrics δ_P and δ_M and the corresponding reference thresholds (Equation 5.6) provide a general indication of whether a software system is suitable for refactoring towards POSP, there are specific cases where refactoring is still beneficial even if these thresholds are not reached. Especially, the thesis suggests the following types of ECUs to be candidates for refactoring due to the consideration of E/E architecture and communication bus constraints:

- *LIN ECUs*: The ECU connected by LIN buses [169] are well-suited for partitioned software due to the relatively low data transmission speed of LIN during software updates. Partitioning the software can significantly reduce the in-line flashing duration.
- *ECUs on congested communication buses*: These ECUs face similar challenges as LIN ECUs, as the bandwidth for data transmission during flashing is insufficient. Additionally, within a CAN network, only two ECUs can be flashed simultaneously, which leads to a bottleneck in the flashing process. For instance, the installation process of S5 in Figure 2.16 could benefit from POSP.

5.2.2 Refactoring software sub-systems

Once the SW-C set is chosen, a refactoring process must be conducted subject to the given boundary conditions. More precisely, a monolithic software system in

the form of S_m must be transitioned into the form of a partitioned software system S_p.

Partitioning a determined SW-C set

Based on Chapter 4, the following constraints must be considered to make the derived partitioned software system usable:

- The basic and customer parts must be independently compilable (by Definition 4.5). Additionally, the basic part cannot cause operational errors on µC-based targets, such as deadlock or watchdog errors.
- The basic part must incorporate the application layer, in which some customer features (see Page 86) are addressed, as illustrated in Figure 4.2.
- The basic part must fulfill design principles derived from usability requirements, such as inter-part communication and arbitration mechanisms, as discussed in Section 4.3.

These constraints do not apply to the design of a monolithic software system, as shown in Figure 5.1-(a). To derive feasible and operational partitioned software systems under these constraints, this thesis suggests a redundancy-based approach that is represented by Line 8 of Algorithm 1, where $R_r()$ denotes the redundancy-based refactoring process for single software set at the application layer. This algorithm partitions a determined software set A_{mi} into A_b and A_{ci} with consideration of the middleware. The middleware is manipulated within the process ($M_{pi} \leftarrow M_{p(i-1)}$).

The transition $R_r()$ shown in Line 8 of Algorithm 1 is detailed in Section 5.3.

We now examine the motivation for introducing redundancy for this transition.

From a software development perspective, redundancy is generally undesirable [170]. However, the introduced redundancy as a result of refactoring is a consequence of the additional constraints imposed by physical separation of the

basic and customer parts. Meeting these newly established software requirements necessitates allocation of additional resources. Consequently, redundancy and increased resource consumption (Section 4.3.4) represent the cost of POSP.

The cost in the form of software redundancy can be explained by the analogy of solving an optimization problem with respect to constraints [9]. Assuming the global optimum has been found for a mathematical optimization problem, adding a constraint can make the found solution infeasible. A similar principle is valid for the design of automotive software systems. The requirements of physically separated basic and customer parts are not met by most of existing automotive software systems, leading to the need of new development or refactoring.

Unlike solving a determined optimization problem that usually has a clear cost function, the quality of software system can hardly be quantified precisely. Therefore, the proposed POSP is a requirement that aims to emphasize the *cost function* of software usage within the end assembly (discussed in Section 2.1.2 and Section A.1). Therefore, one should be aware of the additional costs associated with POSP, avoiding significant increase in the overall *cost function* due to other factors within the software system, such as resource usage or maintenance costs.

Here, the notation of a *cost function* serves as a metaphor for evaluating software quality and trade-offs under constraints. Unlike formal optimization problems where cost functions are explicit, the cost function in software architecture design is more abstract and qualitative. In the context of POSP, a cost function is interpreted as a combination of software metrics, such as memory and computing power usages, development effort, maintainability, and ease of in-line flashing and commissioning.

Handling ECUs with heterogeneous features

When refactoring an ECU with several SW-Cs at the application layer, this thesis recommends an incremental approach, as described in Line 5 - 9 of Algorithm 1 and illustrated in Figure 5.3.

If a considered system S_m is identified as suitable for partitioning and assuming its application layer contains multiple software sub-systems in the form of A_g, namely $A_m = \{A_{m1}, A_{m2}, ..., A_{mn}\}$, the considered software system must be refactored stepwise.

Starting with M_m (the middleware of S_m) as the first input, and a selected software sub-system A_{m1} for the first iteration, the software system S_m can be refactored following the process $R_r()$ outlined in Line 8 of Algorithm 1.

Each time the transformation $R_r()$ is applied, a software sub-system A_{mi} is refactored into the corresponding customer part P_{ci} and the basic part P_b, where P_b contains the modified middleware. Each transformation involves selecting a set of SW-Cs associated with specific features (e.g., power mirrors in the body controller). For the next iteration, the redundancy-based approach $R_r()$ is applied onto next software sub-system A_{m2} with consideration of the altered middleware.

Based on the results of SW-Cs shown in Figure 5.3, the software system can be incrementally refactored, focusing solely on SW-C set A_{m2}. This SW-C set is further refactored into SW-C sets A_{b2} and A_{c2}.

5.2.3 Deployment, verification, and evaluation

Assuming that a determined SW-C set of the considered software system is refactored into partitioned software, for instance,

$$\{M_{p1}, A_{b1}, A_{c1}, A_{m2}, ..., A_{mn}\} = R_r(M_m, A_m) \tag{5.7}$$

the entire software system must be verified before being deployed onto series ECUs. Furthermore, it is assumed that executable files are generated for each partitioned component, such as *basic.hex* for the basic part and *customer1.hex* for customer part 1.

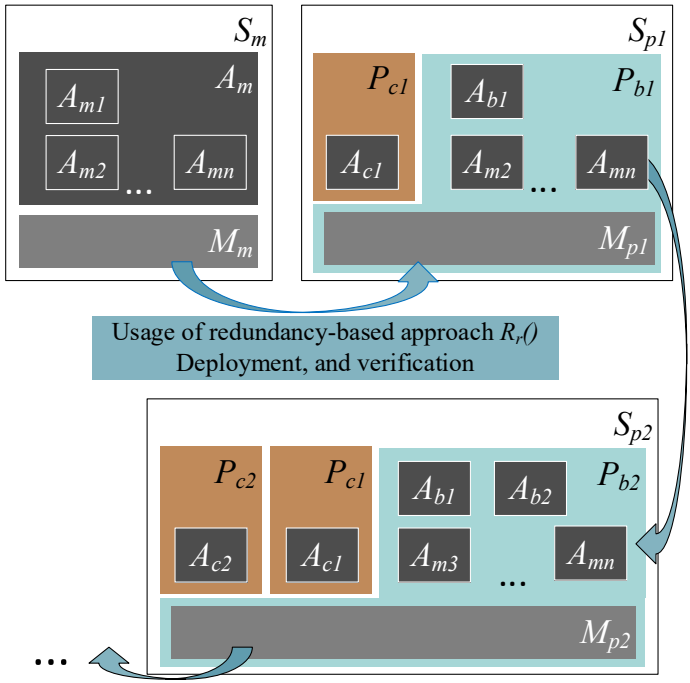


Figure 5.3: The proposed stepwise workflow refactoring software systems according to Algorithm 1.

First, the basic part software including middleware must undergo validation. After the basic part is installed on to the µC-based target, the following steps are suggested using bus simulation:

- *Verification of operability*: The µC-based target with the basic part deployed must function without operational errors, such as dead locks, or improper error codes.
- *Verification of required customer features*: Since the basic part must implement customer features to guarantee the drivability, these implemented customer features must be verified.

- *Verification of diagnostic interfaces*: The diagnostic interfaces used for commissioning and testing must remain unchanged during software refactoring.

Once the basic part software has been verified, customer parts should be installed onto the target. Together with the basic part, customer parts are supposed to provide additional customer features in case the operation conditions are fulfilled, see also Section 4.3.2. After deployment, the following actions regarding verification are suggested:

- *Verification of inter-part communication interfaces*: Ensure that signals are correctly matched and that triggering conditions function as expected.
- *Verification of arbitration mechanisms*: If conflicting signals exist within the software system, the arbitration mechanism must be verified to ensure that the correct output source is selected, as discussed in Section 4.3.2.

For the entire software system, the features implemented by both the basic and customer parts must be validated in a broader environment, at least in interaction with the neighboring ECUs. The unchanged external behaviors indicate the success of the refactoring (described in Line 10 of Algorithm 1), enabling further refactoring of another SW-C set or deployment onto series ECUs. Additionally, in terms of computational performance, the idle time of each µC core should remain below a given threshold (e.g., 90%) after refactoring.

Beyond proving the feasibility of refactoring, the introduced redundancy should also be evaluated. This thesis defines the following metrics:

Definition 5.3: The redundancy factor (r_f)

$$r_f(S_m, S_p) = \frac{s(P_b) + \sum_{i=1}^{n} s(S_{ci})}{s(S_m)} - 1 \tag{5.8}$$

Definition 5.4: The partitioning reduction ratio (δ_{pr})

$$\delta_{pr}(S_m, S_p) = 1 - \frac{s(P_b)}{s(S_m)} \tag{5.9}$$

In which:

- $s(P_b)$ represents the software size of the derived basic part software, measured in kB or MB,
- n denotes the number of customer parts in the derived partitioned software system,
- $s(S_{ci})$ refers to as the size of the i-th customer part,
- $s(S_m)$ stands for the software size of the monolithic software system before refactoring.

We now examine the properties and relevance of these metrics with respect to POSP.

Theorem 1. *For software systems derived from Algorithm 1, the redundancy refactor r_f is bounded below and above. Specifically,*

$$r_f(S_m, S_p) \in (0, 1) \tag{5.10}$$

This theorem indicates that refactoring introduces a certain level of redundancy within the software system. However, the total redundancy introduced cannot exceed the original software size.

Now we prove Theorem 1 regarding the upper and lower limits of r_f for a random software system who is refactored from a monolithic form S_m to the production-oriented partitioned form S_p.

Proof. The software size of the monolithic software system S_m can be represented as the sum of the size of the middleware and the size of its application layer, represented by:

$$s(S_m) = s(M_m) + \sum_{i=1}^{n} s(A_{mi}) \tag{5.11}$$

where $s(M_m)$ is the middleware size of $s(S_m)$, $s(A_{mi})$ denotes the size of the corresponding SW-C set at the application layer of $s(S_m)$.

The partitioned counterpart encompasses the basic and several customer parts, whereas the basic part software contains middleware and its application layer. Therefore, these sizes can be represented by:

$$\begin{aligned} s(S_p) &= s(P_b) + \sum_{i=1}^{n} s(P_{ci}) = s(P_b) + \sum_{i=1}^{n} s(A_{ci}) \\ s(P_b) &= s(M_b) + s(A_b) \end{aligned} \tag{5.12}$$

M_b must realize all features of M_m. Additionally, it must support the realization of P_{ci}, as discussed in Section 4.3. Therefore, its size must be greater than the middleware size of S_m. However, the added software required for physical partitioning cannot surpass twice the middleware size of S_m, Otherwise, implementing it on a newly added µC would be a more viable solution. Therefore,

$$s(M_m) < s(M_b) < 2 \cdot s(M_m) \tag{5.13}$$

As for the size of application layers, the most economical situation is to assign SW-Cs of S_m to P_b and P_{ci}, while the most expensive situation is that P_b and P_{ci} inherits all SW-Cs of S_m. Hence,

$$\sum_{i=1}^{n} s(A_{mi}) \leq s(A_b) + \sum_{i=1}^{n} s(A_{ci}) \leq 2 \sum_{i=1}^{n} s(A_{mi}) \tag{5.14}$$

According to Equation 5.8, the redundancy factor r_f can be derived as follows:

$$
\begin{aligned}
r_f(S_m, S_p) &= \frac{s(P_b) + \sum_{i=1}^{n} s(P_{ci})}{s(S_m)} - 1 && (5.15)\\
&= \frac{s(M_b) + s(A_b) + \sum_{i=1}^{n} s(A_{ci})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} - 1 \\
&> \frac{s(M_m) + s(A_b) + \sum_{i=1}^{n} s(A_{ci})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} - 1 && \textit{(Inequality 5.13)}\\
&\geq \frac{s(M_m) + \sum_{i=1}^{n} s(A_{mi})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} - 1 && \textit{(Inequality 5.14)}\\
&= 0
\end{aligned}
$$

Moreover,

$$
\begin{aligned}
r_f(S_m, S_p) &= \frac{s(M_b) + s(A_b) + \sum_{i=1}^{n} s(A_{ci})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} - 1 && (5.16)\\
&< \frac{2 \cdot s(M_m) + s(A_b) + \sum_{i=1}^{n} s(A_{ci})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} - 1 && \textit{(Inequality 5.13)}\\
&\leq \frac{2 \cdot s(M_m) + 2 \cdot \sum_{i=1}^{n} s(A_{mi})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} - 1 && \textit{(Inequality 5.14)}\\
&= 1
\end{aligned}
$$

Therefore,

$$0 < r_f(S_m, S_p) < 1 \qquad (5.17)$$

□

Furthermore, the effectiveness of the proposed partitioning workflow (Algorithm 1) can be quantitatively estimated under certain assumptions.

Theorem 2. *For a software system S_m, if the production usage ratio $\delta_P(S_m)$ and the middleware ratio $\delta_M(S_m)$ are known, then under the assumptions a_1 - a_3, the production reduction ratio of the considered software system $\delta_{pr}(S_m)$ satisfies*

$$\delta_{pr}(S_m, S_p) \geq \frac{1}{2} \cdot (1 - \delta_P(S_m))(1 - \delta_M(S_m)) \tag{5.18}$$

Assumptions:

(a_1) The number of signals is assumed to proportionally reflect the distribution of functionalities in the compiled software, that is

$$\delta_P(S_m) \approx \frac{\sum_{i=1}^{n} s(A_{mi}) - \sum_{i=1}^{n} s(A_{ci})}{\sum_{i=1}^{n} s(A_{mi})} \tag{5.19}$$

(a_2) The scale of software component overlap is limited to at most half the size of the customer parts derived, specifically:

$$s(A_b) + \frac{1}{2} \sum_{i=1}^{n} s(A_{ci}) \leq \sum_{i=1}^{n} s(A_{mi}) \tag{5.20}$$

(a_3) The increase in middleware size during refactoring is negligible, that is:

$$s(M_b) \approx s(M_m) \approx s(M) \tag{5.21}$$

Theorem 2 shows that the effectiveness of production can be predicted for automotive software systems if proper indicators are chosen, which further implies that all automotive ECUs should undergo the prediction process before being refactored towards POSP.

By applying the threshold values defined in Equation 5.6 in conjunction with Theorem 2, the following minimum required production reduction ratio can be derived:

$$\delta_{prth} = \frac{1}{2} \cdot (1 - 0.3)(1 - 0.7) = 10.5\% \tag{5.22}$$

The proof of Theorem 2 is presented below.

Proof.

$$\begin{aligned} \delta_M(S_m) &= \frac{s(M_m)}{s(S_m)} && (\textit{Equation 5.4}) \\ &= \frac{s(M_m)}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} && (\textit{Equation 5.11}) \end{aligned} \tag{5.23}$$

Therefore,

$$\frac{\sum_{i=1}^{n} s(A_{mi})}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} = 1 - \delta_M(S_m) \tag{5.24}$$

The Inequality 5.20 can be reformulated as

$$s(A_b) \leq \sum_{i=1}^{n} s(A_{mi}) - \frac{1}{2} \cdot \sum_{i=1}^{n} s(A_{ci}) \tag{5.25}$$

Additionally, the approximation shown in Equation 5.19 can be rewritten as follows:

$$\sum_{i=1}^{n} A_{ci} = (1 - \delta_P(S_m)) \sum_{i=1}^{n} s(A_{mi}) \tag{5.26}$$

As a result,

$$
\begin{aligned}
\delta_{pr}(S_m, S_p) &= 1 - \frac{s(P_b)}{s(S_m)} && \textit{(Equation 5.9)} \\
&= 1 - \frac{s(M_b) + s(A_b)}{s(M_m) + \sum_{i=1}^{n} s(A_{mi})} && (\textit{Eq. 5.11 and 5.12}) \\
&\approx \frac{\sum_{i=1}^{n} s(A_{mi}) - s(A_b)}{s(M) + \sum_{i=1}^{n} s(A_{mi})} && (\textit{Approximation 5.21}) \\
&\geq \frac{\sum_{i=1}^{n} s(A_{mi}) - [\sum_{i=1}^{n} s(A_{mi}) - \frac{1}{2}\sum_{i=1}^{n} s(A_{ci})]}{s(M) + \sum_{i=1}^{n} s(A_{mi})} && (\textit{Inequality 5.25}) \\
&= \frac{1}{2} \cdot \frac{\sum_{i=1}^{n} s(A_{ci})}{s(M) + \sum_{i=1}^{n} s(A_{mi})} \\
&= \frac{1}{2} \cdot \frac{(1 - \delta_P(S_m)) \sum_{i=1}^{n} s(A_{mi})}{s(M) + \sum_{i=1}^{n} s(A_{mi})} && (\textit{Equation 5.26}) \\
&= \frac{1}{2} \cdot (1 - \delta_P(S_m))(1 - \delta_M(S_m)) && (\textit{Equation 5.24})
\end{aligned}
$$

□

5.3 Redundancy-based partitioning approach

Based on the introduced generic simplified software model A_g shown in Figure 5.2, the thesis proposes a redundancy-based approach to refactor into production-oriented partitioned software. In the following context, the introduced approach is referred to as Production-Oriented Rundundancy-bAsed Software Partitioning Approach (PORaSPA), denoted as $R_r()$.

The approach PORaSPA corresponds to the transformation of Line 8 in Algorithm 1, see also Section 5.2.2. For a given SW-C set A_m accompanied with the middleware M_m, PORaSPA is expressed as

$$\{P_b, P_c\} = R_r(M_m, A_m) \tag{5.27}$$

where

$$\begin{aligned} P_b &= \{M_b, A_b\} \\ P_c &= \{A_c\} \end{aligned} \tag{5.28}$$

The approach encompasses the following steps:

1. Releasing diagnostic interfaces from application SW-Cs;
2. Duplicating all SW-C except input SW-C;
3. Branching input signals and integrating arbitrators;
4. Reducing overlapped SW-Cs;
5. Assigning SW-Cs and establishing inter-part communication.

which are demonstrated in the following sections and Algorithm 2, where the discussion and derivation mainly focus on the SW-C structure at the application layer, as the middleware configuration and realization rely on the used platform and its vendor.

5.3.1 Releasing diagnostic interfaces from application SW-Cs

The step involves decomposing the application SW-Cs that realize the algorithmic calculations and diagnostic interfaces, denoted as Decompose(W), as shown in Line 2 of Algorithm 2.

Two SW-Cs should be derived from one application SW-C: the one only for the algorithmic functions (represented as W'_A), and the other specifically for processing diagnostic requests, referred to as *diagnostic SW-C* W_D, as shown in Figure 5.4. As diagnostic requests still necessitate information within the application SW-C, interfaces must be created between these two SW-Cs.

For simplicity, this thesis considers the case of a single output SW-C within the examined SW-C set.

Algorithm 2 Realization of PORaSPA

Data: $S_m = \{M_m, A_m\}$, where $A_m = \{W_I, W_A, W_O\}$
Result: $S_p = \{M_p, P_b, P_{c1}\}$,
where $P_b = \{W_I, W_{Db}, W_{Ob}, \alpha\}$
and $P_{ci} = \{W'_{Ac}, W_{Oc}\}$

1: **for** each W_{Ai} in W_A **do**
2: $\{W'_{Ai}, W_{Di}\} \leftarrow$ Decompose (W_{Ai})
3: **end for**
4: $W_{Db} \leftarrow \{W_{D1}, ..., W_{Di}, ...\}$
5: **for** $j = \{A', O\}$ **do**
6: **for** each W_{jk} in W_j **do**
7: $\{W_{j\mathbf{b}k}, W_{j\mathbf{c}k}\} \leftarrow$ Duplicate (W_{jk})
8: **if** Applicable **then**
9: Simplify W_{jbk} and W_{jck}
10: **end if**
11: **end for**
12: Gets α_k as Arbitrator for $\{W_{Obk}, W_{Ock}\}$
13: $\alpha \leftarrow \{\alpha_1, ...\alpha_k, ...\}$
14: $W_{jb} \leftarrow \{W_{jb1}, ..., W_{jbk}, ...\}$
15: $W_{jc} \leftarrow \{W_{jc1}, ..., W_{jck}, ...\}$
16: **end for**
17: **Output** $P_b \leftarrow \{W_I, W_{Db}, W_{Ob}, \alpha\}$, $P_{c1} \leftarrow \{W'_{Ac}, W_{Oc}\}$

The signals *DiagRequest* and *DiagResponse* between W'_A and W_D should only be used, when the internal information within the application SW-C is demanded to be read or written. In case information regarding sensor inputs should be read, a signal channel must be created from input SW-Cs, as shown in Figure 5.4.

In Chapter 4, the thesis proposes to distribute W_D around W'_A, see Section 4.3.3. However, the approach PORaSPA does not recommend to implement application-surrounding diagnostic SW-C within this step, because the refactoring to target application-surrounding diagnostic SW-C involves changes to more signals and makes verification more challenging. Additionally, there is an opportunity to refactor W'_A and W_D within a single software part (e.g., P_b) in later phases.

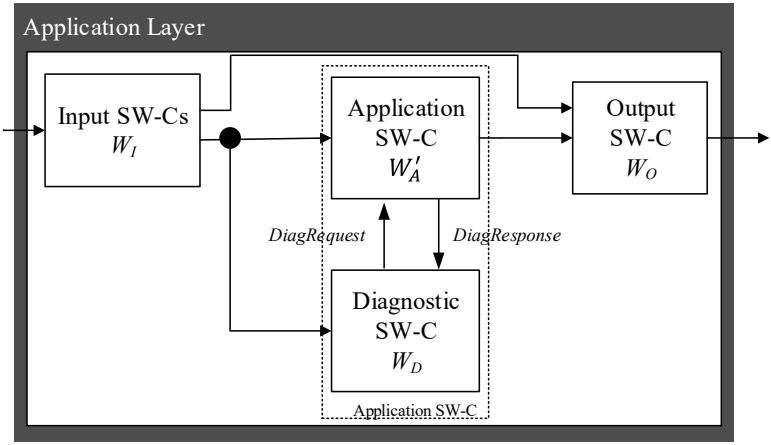


Figure 5.4: S1 - Releasing diagnostic interfaces from application SW-Cs

Compared to the initial state shown in Figure 5.2, this step partially decouples customer and non-customer features related to processing diagnostic requests. The focus here is on restructuring the internal architecture of the application SW-C within the monolithic software system. The newly introduced signal connections and diagnostic SW-C enable subsequent refactoring steps at the software system level without affecting the internal behavior of specific SW-Cs.

5.3.2 Duplicating SW-Cs except input SW-Cs

Based on the decoupled realization of application and diagnostic SW-Cs, the considered set of SW-Cs is now partially duplicated, so that the derived basic part can be compiled and deliver requested customer features, as illustrated in Figure 4.1.

The input signals processed by W_I are strongly dependent on the used sensors and electric designs, which inherently limits the interpretational flexibility of these input SW-Cs. To maintain design flexibility for both the basic and customer-specific parts, all SW-Cs except W_I must be duplicated.

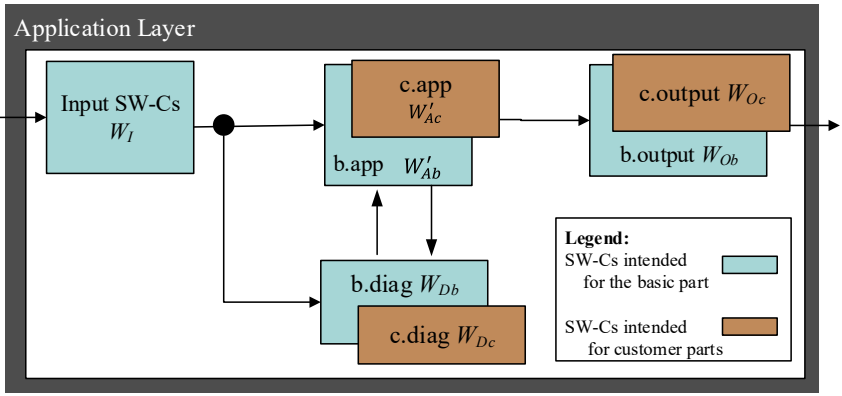


Figure 5.5: S2 - Duplicating all SW-C except input SW-C

The duplicated SW-Cs are assigned to either the basic or customer parts, as illustrated in Figure 5.5 and Line 7 of Algorithm 2. To distinguish SW-Cs from the two parts, the duplicated SW-Cs are labeled with subscripts *b* (for basic) and *c* (for customer). Notably, W_I must remain within the basic part, as the basic part is required to function independently. In contrast, the customer parts can utilize signals provided by the basic part.

Regarding the monolithic software system shown in Figure 5.2, the duplication of SW-Cs necessitates the redefinition of global variables as local variables to ensure uniqueness. Additionally, input signals must be supplied to both copies of the duplicated SW-Cs. However, this approach results in a temporary state that is unsustainable. This is primarily due to the fact that SW-Cs such as *b.output* and *c.output* manipulate the same interface, leading to the loss of compilability and determinism. Moreover, diagnostic requests can be processed by either *b.diag* or *c.diag*, creating ambiguity and rendering the connection to diagnostic-relevant middleware modules infeasible.

5.3.3 Branching input signals and integrating arbitrators

Since the software system shown in Figure 5.5 is neither compilable nor functional, this step aims to restore compilability. To achieve this, the input signals to both application and diagnostic SW-Cs must be branched to ensure that every SW-C requiring the signal receives it. In the context of the general simplified software systems, the signals coming from the input SW-Cs are split into four signals. These signals are then distributed to *c.diag*, *c.user*, *b.user*, and *b.diag*.

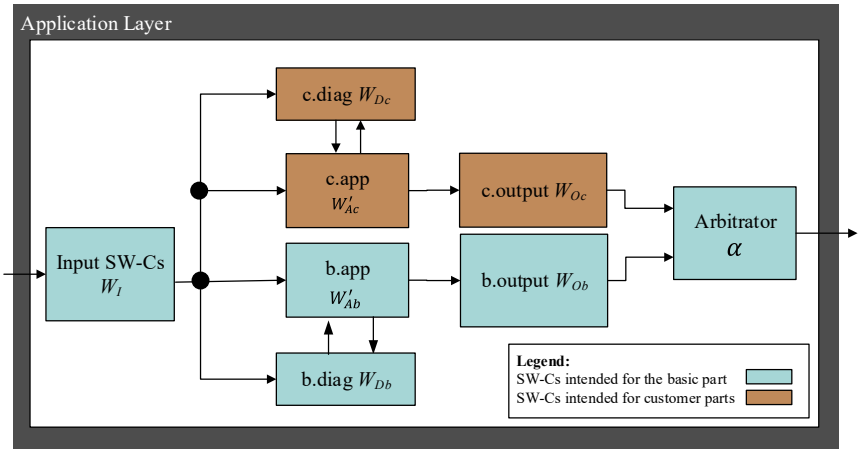


Figure 5.6: S3 - Branching input signals and integrating arbitrators

The conflict arising from output signals is resolved through the integration of arbitrators, illustrated in Figure 5.6 and represented by Line 13 of Algorithm 2. Depending on the type of arbitrator used (as discussed in Section 4.3.2), multiple output signals can be managed by a single arbitrator, if mode variables or state machines are used. Alternatively, each pair of output signals can be assigned to a dedicated arbitrator, particularly when arbitrators with input priorities are employed.

Alongside the necessary adjustments to the middleware, this step restores compilability. Once completed, verification must be conducted (Line 8 of Algorithm 2)

to ensure that the external behaviors remain unchanged when the customer parts are activated.

5.3.4 Reducing overlapped SW-Cs

Now, the software system is compilable, and its external behaviors have been verified to remain unchanged. Furthermore, every SW-C within the considered SW-C set has been appropriately labeled according to its intended software part (basic or customer). Additionally, all SW-Cs, except for the W_I, have been duplicated. Given this context, the focus of software partitioning shifts toward minimizing software overlap while preserving external behaviors and compilability.

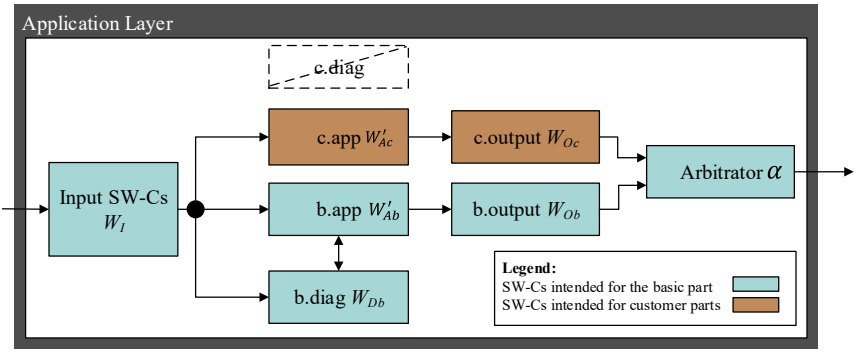


Figure 5.7: S4 - Reducing overlapped SW-Cs

The thesis recommends implementing diagnostic interfaces exclusively within the basic part (as discussed in Section 4.3.3), making the SW-C *c.diag* for the customer part obsolete, as shown in Figure 5.7.

Additionally, redundancy within the duplicated SW-C must be carefully considered, represented in Line 9 of Algorithm 2. For instance, SW-Cs responsible for actuator control can be inherently complex, often structured as SW-C compositions containing multiple SW-Cs for actuator control. However, commissioning and testing procedures within the end assembly typically do not require highly precise

or efficient control algorithms. Instead, production environments prioritize simple control mechanisms and measurable performance from sensors and actuators.

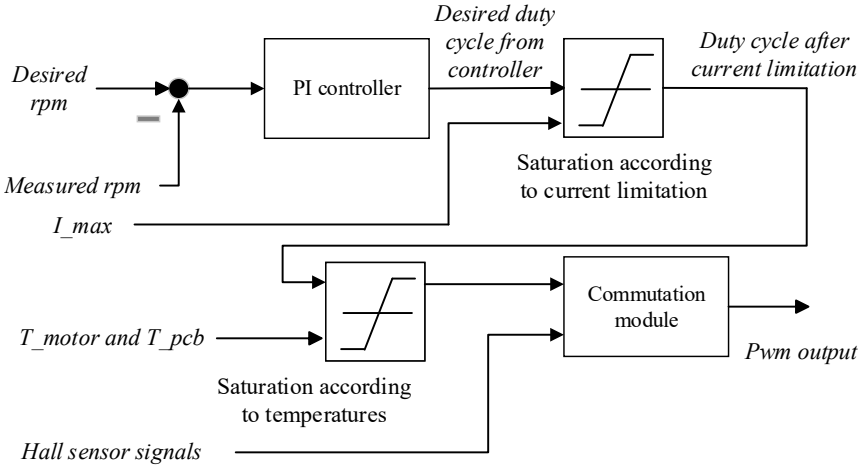


Figure 5.8: Example decomposition of the output SW-C for a BLDC motor controller, denoted as SW_{ob}

To analyze the reduction of redundancy in output SW-Cs, an example software design for a Brushless DC (BLDC) motor controller, denoted as SW_{ob}, is considered, as illustrated in Figure 5.8. The BLDC motor controller is chosen for this analysis due to its widespread adoption in industrial control applications, driven by its high power density, compact size, and straightforward structure [27]. Notably, the SW-C SW_{ob} in Figure 5.7 implements the customer feature of motor control, playing a critical role in determining the overall efficiency of the software system. Additionally, the signal *Pwm output* is often required by diagnostic requests.

The efficiency and safeguarding of motor operation are essential for the operation of customer parts. Therefore, the parameters tuned for the PI controller and the saturation modules in Figure 5.8 are often changed during software alteration. However, the commissioning and testing procedures within the end assembly usually care about the running situations at nominal speed, whose validation only takes a few seconds. Therefore, the two saturation modules shown in Figure 5.8

can be eliminated for P_b (shown in Figure 5.1-(b)). Additionally, the commutation module can be simplified. The variables, such as the lead angle for motor controller [154], can be treated as constants.

This step reduces the overlap caused by SW-C duplication, leading to improved resource utilization and a simplified behavior of the basic part. These changes also enhance the stability of commissioning and testing procedures.

5.3.5 Assigning SW-Cs and establishing inter-part communication

Refactoring on the level of SW-Cs carried out in the illustrated steps facilitates the implementation of physical separation of the considered software system. The implementation of physically separated software systems demands extra considerations compared to monolithic software systems. From the architectural perspective, inter-part communication interfaces must be addressed, as stated in Section 4.3.

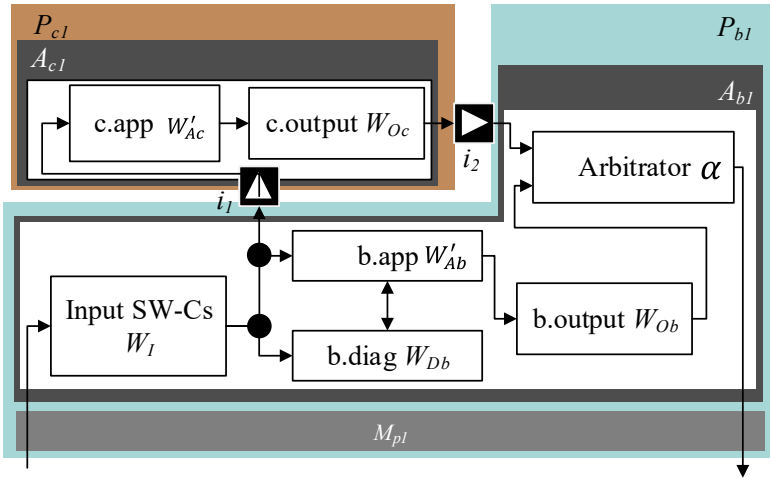


Figure 5.9: S5 - Assigning SW-Cs and establishing inter-part communication, and the partitioned software system S_p is derived

The assignment of SW-Cs should adhere to the principle that only those labeled for customer parts are assigned to them, while the remaining SW-Cs belong to the basic part, as illustrated in Figure 5.9 and Line 14 - 17 of Algorithm 2. Specifically, the input SW-Cs and the arbitrator must reside within the basic part to ensure the feasibility.

Furthermore, P_b must include the middleware, and the distribution of the basic and customer parts must have a reversed L-form. Signals utilized or transmitted by SW-Cs within the customer parts must be managed through inter-part communication interfaces. The interfaces i_1 and i_2, depicted in Figure 5.9, illustrate inter-part communication. Interface i_1 delegates the input signal of A_{c1} from A_b. Since the input signal of A_{c1} is provided by W_I of the basic part, i_1 is categorized as a shared input. In contrast, i_2 is classified as a conflicting signal, as introduced in Section 4.3.1.

5.4 Discussion

5.4.1 Benchmark and trade-offs

In Chapter 4, this thesis presents a concept for partitioning automotive embedded software into basic and customer parts, as defined in Definition 4.6. This chapter presents a workflow to refactor legacy software systems (Algorithm 1) along with a corresponding approach at the application layer (Algorithm 2). While effective, the proposed approach results in redundancies compared to the original monolithic software systems.

The concept and associated approaches aim to address two challenges of automotive software regarding production usage (see also Section 3.5):

- In-line flashing duration, and
- Application-dependent diagnostic interface,

whereas the introduced metric *production reduction ratio* δ_{pr} (Equation 5.9) quantifies how well the partitioned software addresses the aforementioned challenges:

- Assuming that only P_b needs to be installed within the end assembly (usage scenario 2 discussed in Section 4.4.2), and the in-line flashing time is linearly dependent of $s(P_b)$ (Assumption A_1 proposed in Section 4.4.2), δ_{pr} represents the ratio of the reduction in flashing time to the original flashing time.
- Additionally, δ_{pr} reflects the degree of decoupling between the production-irrelevant SW-Cs and P_b. A higher δ_{pr} suggests fewer dependencies, thereby reducing the likelihood that an update to P_{ci} requires changes of P_b.

Empirical observations suggest that a δ_{pr} value exceeding 10% is generally necessary to yield a reduction in in-line flashing time on a minute scale[2], which corresponds to the theoretical value δ_{prth} derived from reference values and Theorem 2 (Equation 5.22).

The desired outcome of applying the software refactoring process toward POSP to the monolithic system S_m^* is a partitioned system S_p^*, such that

$$S_p^* \leftarrow \textit{Algorithm 1}(S_m^*) \tag{5.29}$$

Ideally, this refactored system should satisfy the condition:

$$\begin{aligned} \delta_{pr}(S_m^*, S_p^*) &> 10\% \\ &\approx \delta_{prth} = \frac{1}{2}(1 - \delta_{Mth})(1 - \delta_{Pth}) \end{aligned} \tag{5.30}$$

Beyond the labor costs incurred during the refactoring process, transforming a software system into a production-oriented partitioned form introduces a certain degree of overlap within the software architecture. This redundancy can be

[2] This threshold is an empirical value derived from analysis of production data in 2023.

quantified using the redundancy factor r_f. r_f must remain sufficiently low so that the available computing resources and flash memory of the µC-based system are not exceeded. Furthermore, the benefits of the refactoring, measured by δ_{pr}, should exceed the associated costs measured by r_f:

$$r_f(S_m^*, S_p^*) < \delta_{pr}(S_m^*, S_p^*) \tag{5.31}$$

5.4.2 Benefits and costs

The main benefit of the presented methodology lies in the feasibility. As discussed in Section 3.5.1, application-dependent diagnostic interface is one of the challenges of automotive software in the view of manufacturing. Since production-related features (such as diagnostic interfaces) are integrated within the same SW-Cs as customer features, cutting legacy embedded software systems into the basic and customer parts is impractical, as such a rigid cutting cannot guarantee the compilability and usability of the derived basic part.

Additionally, the illustrated concept offers flexibility for further functional development within the customer parts. New software features can be integrated into customer parts without concerns about production-related constraints, as long as inter-part communication interfaces remain stable. At the same time, this methodology facilitates the implementation of production-relevant software (Definition 2.1) within the basic part.

Taking the software systems S_m and S_p shown in Figure 5.1 as examples to compare software systems before and after software refactoring, the newly requested customer features can be implemented directly within the SW-C without changing the basic part software. As long as the provided inter-part communication interfaces are sufficient to realize the desired customer feature, the considered customer part can be compiled alone and the newly generated binary file for the customer part can replace the former one.

Moreover, the decoupled implementation of diagnostic-relevant features enables target software modifications tailored to the end assembly or after-sales scenarios.

For instance, the SW-C *b.diag* in Figure 5.9 can process assembly-relevant information from diagnostic requests and generate appropriate actuator responses. By leveraging installation status or the vehicle's position within the assembly hall, additional safety constraints to prevent actuator malfunctions can be applied, e.g., deactivating actuators on determined position of assembly hall. This capability mitigates the risk of injury to assembly personnel caused by unintended actuator behavior, see I_2 in Chapter 1, Page 4.

Furthermore, the incremental refactoring ensures the stability of external behaviors of the ECU along the refactoring process, which is important for software systems in large scale.

Nevertheless, a key consideration of the proposed methodology is the redundancy introduced by duplicating SW-Cs within the software system. Software duplication is often debated due to its potential drawbacks, including code bloat, increased memory requirements, and maintenance challenges. Maintaining codes in multiple locations can introduce inconsistencies and complicate updates [170]. In fact, one of the objectives of software refactoring is to minimize code cloning, and various tools have been developed to detect and refactor clones [29]. Especially, overlaps are critical regarding performance and memory usage in µC-based systems.

In this context, redundancy is an inherent consequence of applying additional constraints to the software system (Section 4.3). The duplicated SW-Cs within the basic and customer parts, such as *b.output* and *c.output* in Figure 5.9, serve distinct purposes and cannot replace each other. As a result, while redundancy exists at the level of the entire software system S_p, it becomes irrelevant when considering each software part individually.

Another important consideration is the labor cost and associated risks involved in refactoring of software partitioning. Refactoring must be performed at a detailed level, requiring analysis and categorization of signals, particularly during releasing diagnostic interfaces of application SW-Cs when applying PORaSPA (see Section 5.3.1). Each relevant software module must be carefully examined concerning diagnostic and safety functionalities. Considering that large automotive Simulink models may comprise more than 15,000 blocks [149], and that several thousand

signals are typically recorded via the CAN bus in modern vehicles [56], refactoring according to POSP becomes a highly time-consuming task. Consequently, it demands substantial development effort and incurs high implementation costs.

6 Prototype and Evaluation

Architectural designs are often tied to the platform they are built on, and this is valid for the software architecture of automotive systems, which depend on the capabilities and mechanisms provided by the underlying hardware. As the discussed methodology regarding software partitioning in Chapter 5 brings advantages and disadvantages, it is essential to compare and evaluate the introduced approaches using prototypes, especially in the context of series development.

To demonstrate the effectiveness of the proposed concept illustrated in Chapter 4 in reality, this thesis implements a µC-based software system emulating vehicular door modules according to a formal functional requirement. This system is based on AUTOSAR Classic Platform (CP), ensuring strong relevance to series-production scenarios. Due to the usage of CP, the thesis further utilizes the mechanism Software Cluster Connection (SwCluC) to realize partitioned software within CP-based environment.

A detailed overview to CP is provided in Section A.2, while an analysis of SwCluC and other feasible partitioning mechanisms for µC-based systems are given in Section A.6.

6.1 Prototype design

A pair of software systems (e.g., $\mathscr{S}_x = \{S_{xm}, S_{xp}\}$, cf. Definition 3.1) is considered, where

- S_{xm} is a monolithic software system that can be deployed on the chosen µC-based platform and fulfills all required features.
- S_{xp} is derived by applying *Algorithm 1* to S_{xm} (introduced in Section 5.2), i.e., $S_{xp} \leftarrow$ *Algorithm 1*(S_{xm}).

Then, the introduced concept is feasible, if S_{xp} can also be installed onto the µC platform and provide the same features. Additionally, the impact of the refactoring towards POSP can be evaluated based on the difference between the two software systems using metrics (see also Section A.3 and Section 5.2).

Definition 6.1: System under test

In the following context, the **system under test** refers to the µC-based software system, in which a monolithic software system S_{xm} or its partitioned counterpart S_{xp} is deployed. Both variants, S_{xm} and S_{xp}, provide the same set of customer features.

We now consider requirements for vehicle door modules regarding power windows, who is leveraged to present a production-near software usage, as the door module must be installed, and the power windows must be commissioned and tested within the end assembly, as detailed in Section A.4. These requirements are derived from the formal specifications of a real-world ECU from Mercedes-Benz [69]. They are applied to the system under test to facilitate a comparison between the monolithic and partitioned software systems.

To support the development and validation of the system under test, the relevant usage environments, including both end-user interaction (e.g., vehicle occupants) and production conditions, must be represented. For simplicity in prototype development, these environments are simulated.

A prototype system was built enabling the implementation and verification of a simplified vehicular door module regarding features of power windows. The built prototype system consists of the following modules, as shown in Figure 6.1:

- System under test (Definition 6.1),

- Environment simulator,
- Input simulator, and
- Testing device simulator.

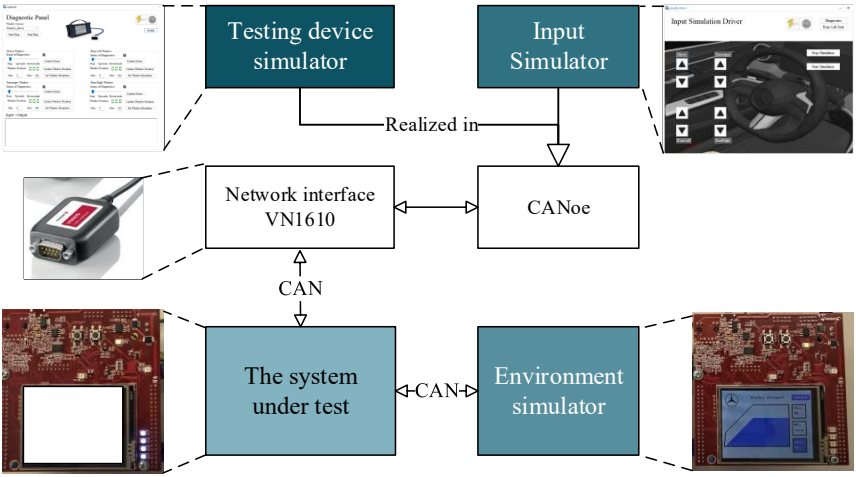


Figure 6.1: Structure of the built prototype

System under test

The system under test consists of an evaluation board platform, on which software that emulates the power window functionalities of a vehicular door module is deployed. Consequently, within the following context, the term *door module* refers to these emulated functionalities realized on the evaluation hardware. The software deployed on the evaluation board is developed specifically for this hardware platform. While the code itself cannot be directly applied in series production, it is based on CP and runs on the same class of µC platform used in concurrent vehicles.

In the context of POSP, within the system under test (implemented on the left board in Figure 6.2), a software system from the considered software system pair $\mathscr{S}_D = \{S_{Dm}, S_{Dp}\}$ must be deployed, where D stands for door modules, m for monolithic system, and p for partitioned system. Both software systems, S_{Dm} and S_{Dp}, are required to satisfy all specified requirements (F_1 - F_4, S_1, S_2, and D_1 - D_5) as outlined in Section A.4.2.

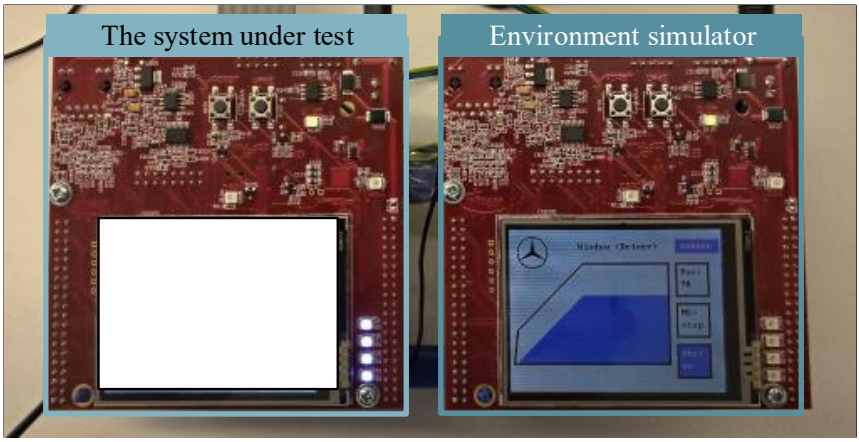


Figure 6.2: Experimental setup with system under test (left) and environment simulator (right)

This thesis leverages AURIX TC3xx 32-bit MCUs [74] to implement the system under test, as this µC family is widely adopted in current automotive ECUs and it supports CP. The software is implemented from scratch with the same development environment and tool chains as series development. Additionally, the usage of partitioning mechanism SwCluC demands a fully functional CP platform, on which additional BSW modules must be configured.

Technically, the system under test is implemented on an evaluation board (3V3 AURIX TC397 Application Kit [76]), where the evaluation board has a TC397 µC as its main controller with several peripherals. The communication between the system under test and the other modules is via CAN bus, where the communication between the system under test and the environment simulator and between the

system under test and the input simulator are configured periodically (once every 100 ms). Besides, the system under test replies to the testing device simulator on demand.

Environment simulator

The environment simulator (demonstrated by the right board in Figure 6.2) acts as a simplified hardware-in-the-loop (HiL) system, enabling the verification and tests of the system under test. More specifically, the environment simulator provides the following functionalities for the prototype:

- Simulation of the behavior of vehicular windows and their surrounding frames,
- Graphical representation of the simulated windows and frames,
- Graphical user interfaces to support testing and verification,
- Reception of motor control signals via CAN and actuation of window movements (aligned with requirements F_1 and F_4 in Section A.4.2),
- Periodic reporting of window positions via CAN, and
- Simulation of obstacles as required by safety requirements S_1 and S_2.

The environment simulator is also implemented using a 3V3 AURIX TC397 Application Kit. Unlike the system under test, which is based on the CP middleware, the environment simulator uses Infineon iLLD (Low-Level Driver) framework [75]. This design choice allows for the usage of the LCD display on the evaluation board (visible in Figure 6.2). Specifically, the LCD display on the right board provides a real-time state of a simulated window, featuring a black frame representing the window frame and a blue area for the glass. Adjacent to the graphic are three data fields displaying the window's current position (0-255), the motor's operational status, and obstacle detection. The touchscreen capability allows for configuration changes.

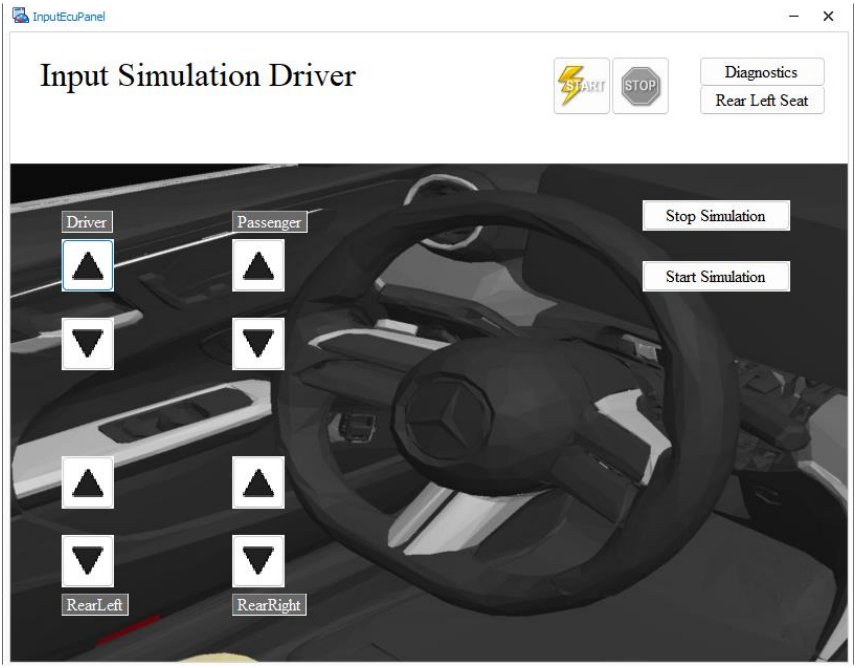


Figure 6.3: Input simulator for driver seat button interactions

Input Simulator

The input simulator, illustrated in Figure 6.3, emulates user interactions by simulating vehicle occupants pressing window control buttons. It is implemented using CANoe [161], with a dedicated user interface developed via CANoe Panels.

As CANoe provides PC-based bus simulation capabilities, the system under test is connected to the input simulator through a VN1610 network interface [163].

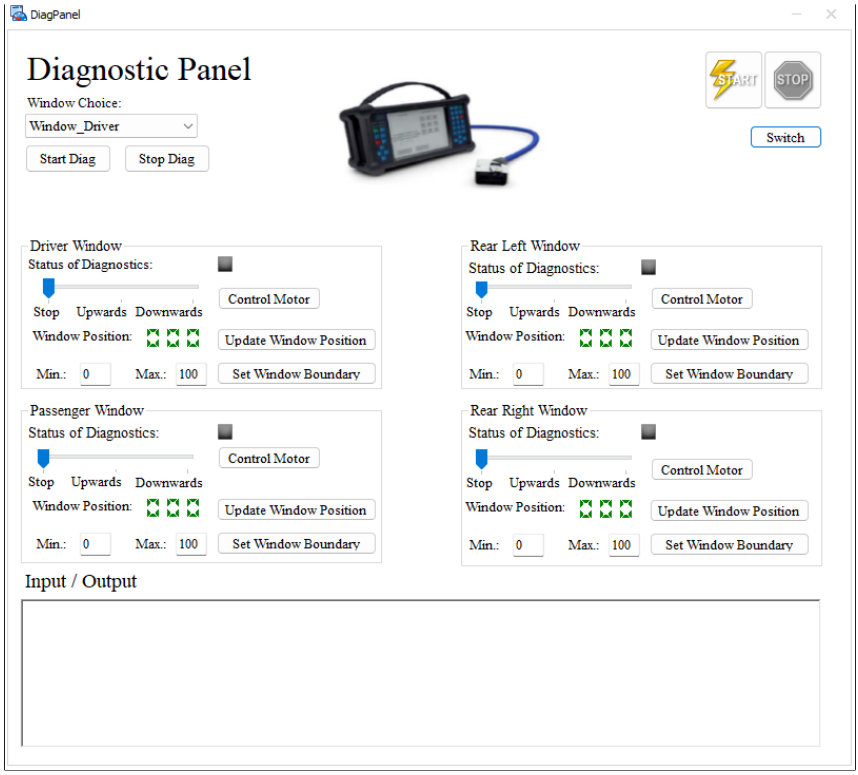


Figure 6.4: Testing device simulator

Testing device simulator

The testing device simulator is also realized with the usage of CANoe, sharing the same configuration with the input simulator. However, the testing device simulator has its own Panel, as shown in Figure 6.4.

The testing device communicates with the system under test using Unified Diagnostics Services (UDS) protocol, providing the following features for the prototype system:

- Entering and exiting the diagnostic session,
- Changing the driving status of motors,
- Acquiring the current window position for the corresponding window, and
- Writing the window boundary positions.

6.2 Software architectures in the system under test

This section analyzes the software system pair $\mathscr{S}_D = \{S_{Dm}, S_{Dp}\}$ within the system under test, characterized as follows:

1. Both systems implement all the required functionalities, namely F_1 - F_4, S_1, S_2, and D_1 - D_5 (Section A.4.2) of the vehicular door modules at driver and co-driver seats.
2. Both systems are deployable on the chosen software and hardware platform (CP, AURIX TC397).
3. S_{Dp} is derived from S_{Dm} using the proposed concept and methodology outlined in Chapter 4 and Chapter 5.
4. $S_{Dp} = \{P_{Db}, P_{Dc}\}$, in which P_{Db} represents the basic part and P_{Dc} represents the customer part. Note that the considered partitioned system S_{Dp} has only one customer part.

6.2.1 The monolithic software system

Analogous to Equation 5.1 and the depiction in Figure 5.1, the monolithic system S_{Dm} is composed of an application layer and middleware:

$$S_{Dm} = \{A_{Dm}, M_{Dm}\} \tag{6.1}$$

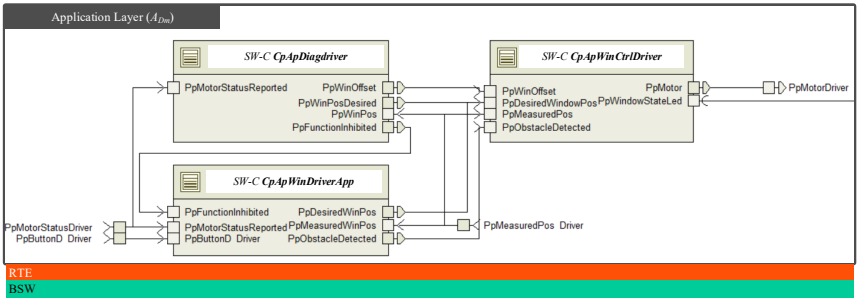


Figure 6.5: Excerpt of the software design for the monolithic system S_{Dm} in DaVinci Developer [159], focusing on the SW-Cs that implement the driver-side functionalities.

At the application layer A_{Dm}, the window control functionalities for a single window are realized by three SW-Cs. Figure 6.5 shows the software architecture for the window at driver seat, consisting of:

- *CpApDiagDriver* – a diagnostic SW-C,
- *CpApWinCtrlDriver* – an output SW-C, and
- *CpApWinDriverApp* – an application SW-C.

The application layer can be represented as:

$$A_{Dm} = \{\textit{CpAp}\boldsymbol{x}\textit{Driver, CpAp}\boldsymbol{x}\textit{Passenger}\} \tag{6.2}$$

where the placeholder x represents the specific type of SW-Cs.

The SW-Cs implementing the passenger-side functionalities differ slightly, as the window at other seats can be influenced by two buttons (a button at driver seat and a button for itself). The diagnostic and output SW-Cs for other windows can be instantiated from the driver-side SW-C with appropriate external signal connections.

The following discussion focuses exclusively on the SW-Cs associated with the driver's window, as SW-Cs for windows at other occupants can be implemented in a similar way and are feasible within the same BSW configuration.

Compared to the generic simplified model at the application layer A_g (illustrated in Figure 5.2), the application layer A_{Dm} does not include an explicit input SW-C. Additionally, the diagnostic and application SW-Cs are designed as decoupled components. This decoupling emphasizes the usage of diagnostic interfaces, avoiding the discussion of releasing diagnostic interfaces from application SW-Cs during software refactoring, see also Figure 5.4.

The software system S_{Dm} can be firstly evaluated using the production usage ratio as defined in Equation 5.3:

$$\delta_P(S_{Dm}) = \frac{|D(A_{Dm})|}{|I(A_{Dm})| + |O(A_{Dm})|} = \frac{16}{43} = 0.37 \tag{6.3}$$

which complies with the specified reference threshold given in Equation 5.6.

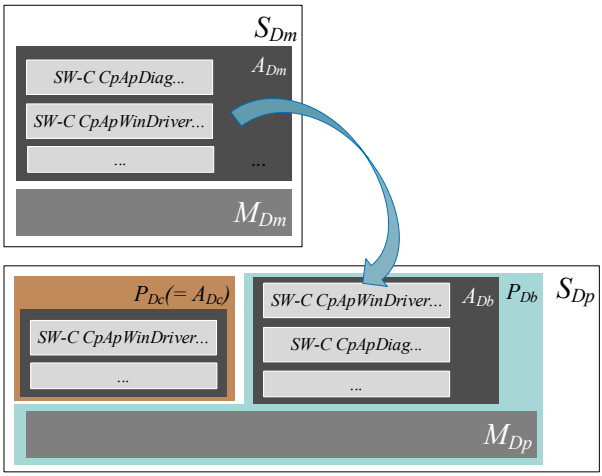


Figure 6.6: Refactoring S_{Dm} based on Algorithm 1, cf. Figure 5.3.

6.2.2 The partitioned software system

Applying the workflow described in Algorithm 1, S_{Dm} is refactored into a partitioned form S_{Dp} as illustrated in Figure 6.6, where

$$\begin{aligned} S_{Dp} &= \{P_{Db}, P_{Dc}\} \\ P_{Db} &= \{A_{Db}, M_{Db}\} \\ P_{Dc} &= A_{Dc} \end{aligned} \tag{6.4}$$

The derived basic part P_{Db} corresponds to the *Host Cluster*, while the derived customer part P_{Dc} is represented by an *Applicative Cluster* within the mechanism SwCluC [15], see also Section A.6.2.

At the application layer, the inter-part communication interfaces are established as illustrated in Figure 6.7. These interfaces span across software parts, serving as boundary conditions for P_{Db} and P_{Dc}.

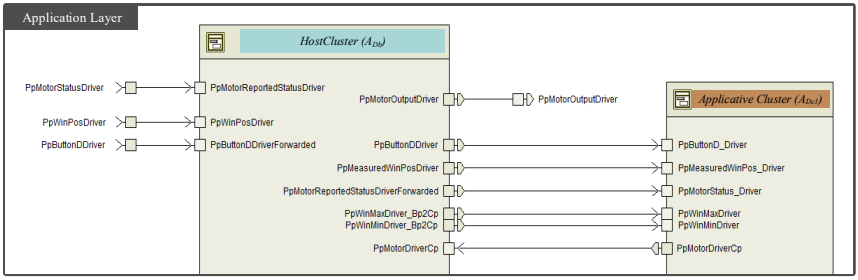


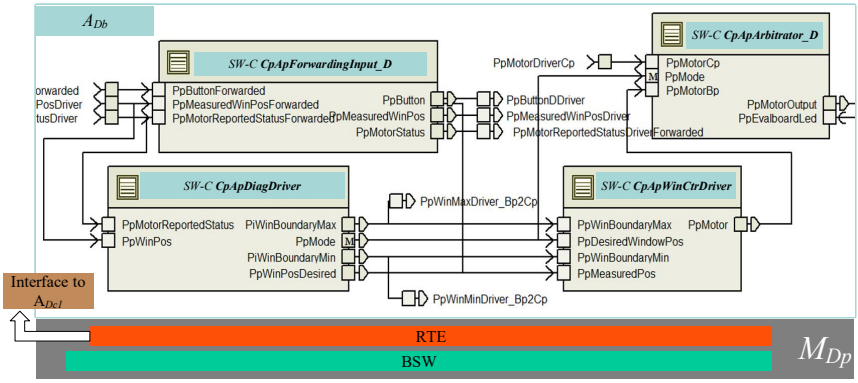
Figure 6.7: Overview of the partitioned system S_{Dp} at the application layer (Project *SystemArchitect* in MICROSAR)

Within P_{Db}, its application layer A_{Db} (illustrated in Figure 6.8-(a)) consists of four SW-Cs, in which:

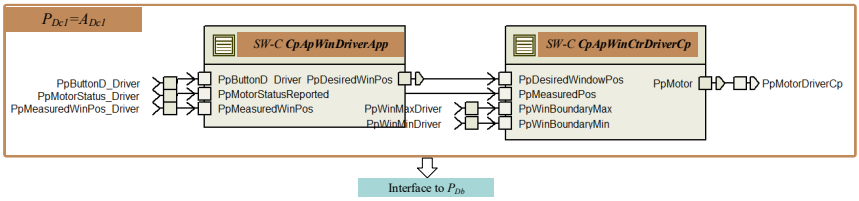
- *CpApForwardingInput_D*: This SW-C does not provide any functional behavior but serves as a repeater of shared inputs for P_{Dc}. Such forwarding

is necessary because intra-part signals cannot be directly exposed as inter-part communication interfaces. This SW-C corresponds to Line 7 of Algorithm 2.

- *CpApDiagDriver*: This SW-C inherits the SW-C *CpApDiagDriver* of A_{Dm} (see Figure 6.7).
- *CpApArbitrator_D*: This SW-C implements an arbitration mechanism to resolve the conflicting output *PpMotorOutputDriver*, corresponding to Line 12 and 13 of Algorithm 2.
- *CpApWinCtrDriver*: This SW-C is a simplified version of SW-C *CpApWinctrlDriver* of A_{Dm}, corresponding to Line 9 of Algorithm 2.



(a) SW-Cs of P_{Db}



(b) SW-Cs of P_{Dc}

Figure 6.8: Excerpt of the software design for of P_{Db} and P_{Dc} in DaVinci Developer [159], focusing on SW-Cs implementing the driver-side functionalities, cf. the depiction of the software refactoring process shown in Figure 6.6.

The customer part ($P_{Dc} = A_{Dc}$) has two SW-Cs for the driver-side window where

- *CpApWinDriverApp*: inherits the SW-C of the same notation of A_{Dm}.
- *CpApWinCtrDriverCp*: stands as the counterpart of SW-C *CpApWinCtrDriver* in P_{Db}, whereas the focus and logic of this SW-C differ from its counterpart in A_{Db}. For instance, only the SW-C in P_{Dc} takes anti-trap protection into consideration.

6.3 Evaluation of implemented software systems

The software systems S_{Dm} and S_{Dp} were successfully implemented onto the system under test and fulfill all specified requirements. Specifically, within S_{Dp}:

- P_{Db} satisfies all diagnostic-relevant requirements (D_1 - D_5 outlined in Section A.4.2), if the arbitrators (e.g., SW-C *CpApArbitrator_D* shown in Figure 6.7) deactivate the customer part P_{Dc}.
- Based on the middleware and inter-part communication interfaces provided by P_{Db}, P_{Dc} satisfies all customer and safety requirements (F_1 - F_4, S_1, and S_2), if the arbitrators activate the customer part functionalities.

The feasibility of S_{Dp} demonstrates the usability of POSP. Additionally, the usage of CP and SwCluC demonstrates the practical relevance for industrial applications, particularly for µC-based automotive software systems.

6.3.1 Evaluation of installability

As discussed in Section 5.4, the metric δ_{pr} quantifies how effectively a monolithic software system is partitioned with the presented methodology. More specifically, δ_{pr} represents the potential time savings for in-line flashing, if only the basic part

is installed during the end assembly, see also Figure 4.11. As a result, δ_{pr} serves as an installability improvement indicator. The software sizes of each layer in the system pair $\mathscr{S}_D = \{S_{Dm}, S_{Dp}\}$ is now analyzed with particular focus on the sizes of middleware and SW-Cs at the application layer.

For compilable systems S_{Dm}, P_{Db}, and P_{Dc}, their sizes $s(S_{Dm})$, $s(P_{Db})$, and $s(P_{Dc})$ are directly measurable. Middleware size ($s(M_x)$) is determined by removing all SW-Cs at the application layer and recompiling, while the application layer size $s(A_x)$ is derived from:

$$s(A_x) = s(S_x) - s(M_x) \tag{6.5}$$

in which x is the placeholder for either Dm or Db.

As a result, sizes of every layer in systems of software system pair $\mathscr{S}_D$ are presented in Table 6.1.

Table 6.1: Software sizes of the considered software system pair $\mathscr{S}_D$

S_{Dm}		S_{Dp}	
Size of	Size in Byte	Size of	Size in Byte
S_{Dm}	421.592	P_{Db}	424.340
M_{Dm}	405.212	M_{Dp}	410.623
A_{Dm}	16.380	A_{Db}	13.717
		P_{Dc}	7.821

In the case of the partitioned software system $S_{Dp} = \{P_{Db}, P_{Dc}\}$, the relevant software sizes are those of the basic part P_{Db} and the customer part P_{Dc}. As the middleware of P_{Dc} is excluded from consideration, its size directly corresponds to that of the application layer, namely

$$s(P_{Dc}) = s(A_{Dc}) = 7{,}821 \textit{ Byte}$$

The compiled size of the basic part amounts to 424,340 Byte. By removing all SW-Cs at the application layer and re-compiling, the middleware size is obtained as $s(M_{Dp}) = 410{,}623$ Byte. Consequently, the size of the application layer in the basic part is derived as the difference $s(P_{Db}) - s(M_{Dp}) = 13{,}717$ Byte.

Before software refactoring towards production-oriented partitioned software, this thesis proposes to pre-check the characteristics of the considered system. However, the pre-check is not meaningful for software on the developed prototype, as the utilized middleware stack (Vector MICROSAR) is industry-level and much more comprehensive than the required features listed in Section A.4.2. For instance, the middleware ratio of δ_M (Equation 5.4) of S_{Dm} can be calculated as follows:

$$\delta_M(S_{Dm}) = \frac{s(M_{Dm})}{s(S_{Dm})} = 96\% \tag{6.6}$$

While pre-checking system characteristics is theoretically advisable, the prototype's industrial-grade Vector MICROSAR middleware (with $\delta_M(S_{Dm}) = 96\%$) makes the pre-checking regarding the middleware ratio impractical for the case study. This value significantly exceeds the threshold in Equation 5.22, suggesting S_{Dm} is suboptimal for production-oriented partitioning. However, the prototype serves to validate conceptual feasibility rather than series production readiness.

In series production ECUs, the middleware typically constitutes about 15%[1] of the total software size according to internal observation. For example, a series ECU can have a generated binary file for about 25 MB, within which the middleware size is typically limited to 4 MB. Applying this assumption to S_{Dm} and P_{Db} yields:

$$\delta_M(S_x) = \frac{s(M_x)}{s(S_x)} \approx 0.15 \tag{6.7}$$

[1] This figure is the arithmetic mean derived from a static analysis of the software memory distribution across a about 30 series-production ECUs.

Based on Equation 6.5, Equation 6.7 can be rewritten as:

$$\frac{s(M_x)}{s(A_x)} \approx 0.19 \tag{6.8}$$

where it is valid to replace x by Db or Dm.

This implies that SW-Cs at the application layer dominate software size, mitigating evaluation bias deduced by BSW. Consequently:

$$\begin{aligned} s(S_{Dm}) &= s(M_{Dm}) + s(A_{Dm}) \approx 1.19 \cdot s(A_{Dm}) \\ s(P_{Db}) &= s(M_{Db}) + s(A_{Db}) \approx 1.19 \cdot s(A_{Db}) \end{aligned} \tag{6.9}$$

Using the derived sizes of S_{Dm} and P_{Db} in Equation 6.9, the redundancy factor r_f (defined in Equation 5.8) and the production reduction ratio δ_{pr} (defined in Equation 5.9) of $\mathscr{S}_D$ are derived as follows:

$$\begin{aligned} r_f(S_{Dp}, S_{Dm}) &= \frac{s(P_{Db}) + s(P_{Dc})}{s(S_{Dm})} - 1 \\ &\approx \frac{1.19 \cdot s(A_{Db}) + s(P_{Dc})}{1.19 \cdot s(A_{Dm})} - 1 \\ &= 24\% \end{aligned} \tag{6.10}$$

$$\begin{aligned} \delta_{pr}(S_{Dp}, S_{Dm}) &= 1 - \frac{s(P_{Db})}{s(S_{Dm})} \\ &\approx 1 - \frac{1.19 \cdot s(A_{Db})}{1.19 \cdot s(A_{Dm})} \\ &= 16\% \end{aligned} \tag{6.11}$$

These results show that partitioning reduces the software size for in-line flashing by 16%, if only the basic part needs to be installed within the end assembly, as shown in Figure 4.11-(b). However, the refactoring incurs 24% memory redundancy based on these results. While Theorem 1 is validated ($\delta_{pr} \in (0, 1)$), Theorem 2 remains unverified due to application of assumption showed in Equation 6.7. Although δ_{pr}

meets the criterion shown in Inequality 5.30, the cost-benefit imbalance ($\delta_{pr} < r_f$) reveals suboptimal trade-offs due to:

- *Mandatory SW-C duplication (Algorithm 2)*: The duplication is compulsory for compilability but increases r_f without δ_{pr} gains. However, the duplicated SW-Cs provide flexibility in long term, as these SW-Cs in every part allow for further modification.
- *Simplified functional scope*: The functional scope of the software system considered (illustrated in Section A.4) is significantly simplified in comparison with the version for real-world implementations, which leads to the ratio of application SW-Cs lower than it in real-world practice. Therefore, a large number of of SW-Cs in A_{Dm} must be redundantly allocated in A_{Db} and A_{Dc}, leading to a lower δ_{pr}.
- *Inter-partition communication*: Signal forwarding SW-Cs (e.g., *CpApForwardingInput_D* in Figure 6.8-(a)) demand additional resources.

6.3.2 Evaluation of code structure

Beyond the evaluation of installability associated with δ_{pr}, the software pair $\mathscr{S}_D$ is also assessed in terms of its structural characteristics using quantitative code metrics. Since the majority of the source code consists of generated code for BSW and RTE, the discussion on lines of code (Section A.3) is not meaningful. Therefore, this thesis focuses on McCabe's Cyclomatic Complexity (M_{cc}) (Equation A.12) as a structural metric at the application layer.

As M_{cc} is measured at the function or Runnable level, this thesis evaluates the structural characteristics of SW-Cs based on the aggregated M_{cc} values of their contained Runnables, denoted by $\sum M_{cc}$, as well as the number of software entities (SW-Cs and Runnables) at each level. Notably, SW-Cs that serve solely for arbitration purposes, such as *CpApArbitrator_D* in Figure 6.8-(a), are excluded from the analysis. Furthermore, only Runnables with a high degree of complication

($M_{cc} > 5$) are considered, as these Runnables typically implement application algorithms.

Table 6.2: Key complexity metrics across implemented software systems $\mathscr{S}_D$

Characteristics	S_{Dm}	S_{Dp}	P_{Db}	P_{Dc}
Number of SW-Cs	3	5	3	2
Number of Runnables	21	28	19	9
$\sum M_{cc}$	34	44	10	34

Table 6.2 summarizes the evaluation results of the software pair $\mathscr{S}_D$. A direct comparison between the system before and after refactoring (S_{Dm} and S_{Dp}) reveals that the refactoring increases structural load across all considered dimensions: number of SW-Cs, Runnables, and $\sum M_{cc}$, as expressed by:

$$\sum M_{cc}(S_{Dm}) < \sum M_{cc}(S_{Dp}) \tag{6.12}$$

However, the derived software partitions P_{Db} and P_{Dc} show a re-distribution of structure. Although P_{Db} retains the same number of SW-Cs as S_{Dm}, its $\sum M_{cc}$ is significantly lower. Conversely, P_{Dc} inherits the structural weight in terms of $\sum M_{cc}$, but with fewer SW-Cs and Runnables. This indicates that the POSP effectively decouples structural responsibilities, with structural load at the SW-C level retained in P_{Db}:

$$|\text{SW-C}(S_{Dm})| = |\text{SW-C}(P_{Db})| \tag{6.13}$$

and structural load at the Runnable level shifted into P_{Dc}:

$$\sum M_{cc}(S_{Dm}) = \sum M_{cc}(P_{Dc}) \tag{6.14}$$

This distribution is considered desirable. Concentrating architectural entities (measured by number of SW-Cs) in the basic part aligns with the goal of maintaining a stable architecture throughout the software life cycle. Moreover, allocating

dynamic, implementation-heavy logic (measured by $\sum M_{cc}$) to customer parts allows for flexibility in frequent updates, such as feature enhancements and bug fixes.

6.3.3 Evaluation of flexibility

One of the key challenges of automotive software from the view of manufacturing is the *application-dependent diagnostic interface* (see Section 3.5.1). This dependency causes the phenomenon that software updates performed after Start of Production (SOP) can result in failures during commissioning and testing at the end of the assembly line. Consequently, the flexibility of the software systems considered with respect to frequent and regular software updates is an additional evaluation criterion for software refactoring.

For the software system pair $\mathscr{S}_D$, the following three update scenarios are considered:

(U_1) **Functional updates & improvements**: modify software features without changing the communication matrix or software architecture at the application layer.

Example: A new control algorithm is designed for the SW-C *CpApWinCtrDriver* in Figure 6.5 to improve the efficiency of the motors driving vehicle windows.

(U_2) **Updates to diagnostic interfaces**: adapt or extend diagnostic interfaces to enable new commissioning or testing procedures.

Example: A new testing strategy requires real-time monitoring of motor output, which necessitates modifications to the diagnostic interfaces of the corresponding SW-C such as *CpApWinDriverApp* shown in Figure 6.5.

(U_3) **Introduction of new customer features**: addresses newly requested features from changed requirements.

Example: A feature is demanded that automatically closes the windows when the vehicle velocity exceeds a threshold (e.g., 30 km/h). Although the vehicle velocity signal is distributed within the E/E topology, it is not available to the software systems S_{Dm} or S_{Dp}, as illustrated in Figure 6.5 and Figure 6.8.

The two software systems S_{Dm} and S_{Dp} are evaluated the three update scenarios described above.

In the monolithic system S_{Dm}, U_1 involves altering application SW-Cs like *CpApWinCtrDriver* shown in Figure 6.5, followed by re-generation of BSW/RTE, re-compilation, and re-validation. U_2 requires updating diagnostic SW-Cs (e.g., *CpApDiagDriver* in Figure 6.5) and re-compilation. U_3 demands changes to the communication matrix and application SW-Cs, again triggering re-generation, re-compilation, and re-validation.

For the partitioned system S_{Dp}, the impacts are more localized. U_1 necessitates changes within P_{Dc} (e.g., *CpApWinDriverApp* shown in Figure 6.8-(b)), requiring re-compilation and re-validation of P_{Dc}. U_2 is handled exclusively by P_{Db} (via *CpApDiagDriver* shown in Figure 6.8-(a)), leaving P_{Dc} intact. U_3 requires updates to P_{Dc} and the communication matrix, leading to BSW/RTE regeneration in P_{Db}, re-compilation of both parts, and full system re-validation.

Table 6.3 summarizes impacts of the presented update scenarios on both software systems. Based on the comparison, the following statements are derived:

- Partitioned software systems based on POSP gain flexibility when changes can be constrained to a specific software part. For instance, production-irrelevant functional updates require modifications only to the affected customer parts, eliminating the need to re-generate middleware or re-validate the entire system.
- However, this flexibility is lost when changes affect the system's boundary conditions, such as alterations to the communication matrix that require updates to inter-part communication interfaces. In such cases, the middleware

Table 6.3: Impact comparison of update scenarios on S_{Dm} and S_{Dp}

Scenario	**Impact on S_{Dm}**	**Impact on S_{Dp}**
U_1	Alteration of application SW-Cs; Regeneration of middleware; re-compilation; re-validation of the whole software system	Alteration of application SW-Cs within P_{Dc}; re-compilation; re-validation of P_{Dc}.
U_2	Alteration of diagnostic-relevant SW-Cs; re-compilation; re-validation of S_{Dm}	Update of diagnostic-relevant SW-C in P_{Db}, re-compilation; re-validation of P_{Db}
U_3	Update of application SW-Cs; re-generation of middleware; re-compilation; re-validation.	Update of SW-Cs in P_{Dc}; re-generation of middleware in P_{Db}; re-compilation of both P_{Db} and P_{Dc}; re-validation of the whole system.

in P_{Db} must be re-generated, necessitating the re-validation of P_{Db}, P_{Dc}, and their integration.

Towards a broader evaluation of POSP

The developed prototype addresses a single use case of vehicular power window functionalities, which serves as representative but does not fully reflect the diversity of automotive software systems deployed across various ECUs. To enable a broader analysis of POSP, this thesis extends the design and implementation of S_{Dm} in two directions:

1. By expanding the set of functionalities required from S_{Dm} (discussed in Section 6.4);
2. By considering different variants of S_{Dm} (outlined in Section 6.5).

6.4 Implications of increased functional scope

To improve the informative value of the developed prototype, the software system pair considered $\mathscr{S}_D$ is extended to $\mathscr{S}'_D = \{S'_{Dm}, S'_{Dp}\}$ where $\mathscr{S}'_D$ implements the control of four windows. In contrast, $\mathscr{S}_D$ only implements the functionalities of the two front windows.

In the extended context, $\mathscr{S}_D$ and $\mathscr{S}'_D$ shares the same middleware for the corresponding software system, described as:

$$\begin{aligned} S'_{Dm} &= \{M_{Dm}, A'_{Dm}\} \\ S'_{Dp} &= \{P'_{Db}, P'_{Dc}\} \\ P'_{Db} &= \{M_{Dp}, A'_{Db}\} \\ P'_{Dc} &= A'_{Dc} \end{aligned} \tag{6.15}$$

The compilation of the extended software system pair leads to the corresponding software sizes of all considered software layers, shown in Table 6.4.

Table 6.4: Software sizes of the software system pair considered $\mathscr{S}'_D$

S'_{Dm}		S'_{Dp}	
Size of	Size in Byte	Size of	Size in Byte
S'_{Dm}	430.776	P'_{Db}	432.810
M_{Dm}	405.212	M_{Dp}	410.623
A'_{Dm}	25.564	A'_{Db}	22.187
		P'_{Dc}	11.541

The comparison between software sizes of the software systems considered within $\mathscr{S}_D$ and $\mathscr{S}'_D$ presents the fact that, despite the functional expansion from two to four windows, the size of the application layers does not increase proportionally. This is because certain SW-Cs supporting similar functionalities are reused through multiple instantiation. For example, the component *CpApWinCtrDriver* shown in

Figure 6.8 is instantiated multiple times to handle rear window control without full duplicating the implementation.

Using the same metric definitions (the redundancy factor r_f in Equation 5.8, the production reduction ratio δ_{pr} in Equation 5.9, and the assumed middleware ratio presented in Equation 6.8), the relevant metrics of $\mathscr{S}'_D$ are derived as:

$$\begin{aligned} r_f(S'_{Dp}, S'_{Dm}) &= 25\% \\ \delta_{pr}(S'_{Dp}, S'_{Dm}) &= 13\% \end{aligned} \tag{6.16}$$

A comparative analysis between the evaluations of $\mathscr{S}_D$ (from Equations 6.10 and 6.11) and those of $\mathscr{S}'_D$ (Equation 6.16) leads to the following facts:

- Redundancy factors of both software system pairs are similar:

  $$r_f(S'_{Dp}, S'_{Dm}) \approx r_f(S_{Dp}, S_{Dm}) \tag{6.17}$$

  which indicates that linear expanding software systems does not increase the additional memory overhead induced by the software partitioning.

- The production reduction ratio of the extended system shrinks in spite of linear expansion of implemented functionalities, described as

  $$\delta_{pr}(S'_{Dp}, S'_{Dm}) < \delta_{pr}(S_{Dp}, S_{Dm}) \tag{6.18}$$

  where the reason behind the fact can be explained by the following factors:

  - Expansion of implemented functionalities does not lead to corresponding expansion of production-irrelevant applications that are outsourced to customer parts due to the multiply instantiated SW-Cs within the customer part which saves the memory usage.
  - However, the expanded functionality results in corresponding expansion of middleware, primarily due to the additional intra-part and inter-part communication interfaces.

6.5 Handling software variants in partitioned systems

As mentioned in Section 4.4, usage of software partitioning can lead to an increase in software entities to be managed. Instead of dealing with a single software package for an ECU, it becomes necessary to manage the basic and customer parts within the software system. Additionally, the frequently evolving software leads to exponentially increasing numbers of software versions and variants [138], see also the detailed discussion given in Section A.5.

Modern automotive software systems are not only time-variant but also characterized by a high degree of variability. Given the added variant richness introduced by POSP, see also Section 4.4, it is crucial to analyze the proposed concept and methodology within the context of variant-rich systems. Such an analysis will ensure the applicability of the approach, enabling it to address both temporal and structural variations in µC-based automotive software systems.

For the analysis of variability regarding POSP, this thesis extends the example software system S_{Dm} and its partitioned counterpart S_{Dp}, whose requirements are presented in Section A.4. Using the workflow shown in Algorithm 1 in Chapter 5, the software system S_{Dm} is refactored to partitioned form S_{Dp}. The analysis of POSP is based on the comparison of S_{Dm} and S_{Dp} with respect to the variability realization mechanisms.

6.5.1 Software design with variants

The design for the software system pair $\mathscr{S}_D$ is now considered with a fine-grained realization at the SW-C level, so that the influence of variant richness is illustrated more directly within the architecture. Besides the output SW-C *Motor Controller*, several input SW-Cs are used for the software design regarding variant-richness analysis, as illustrated in Figure 6.9.

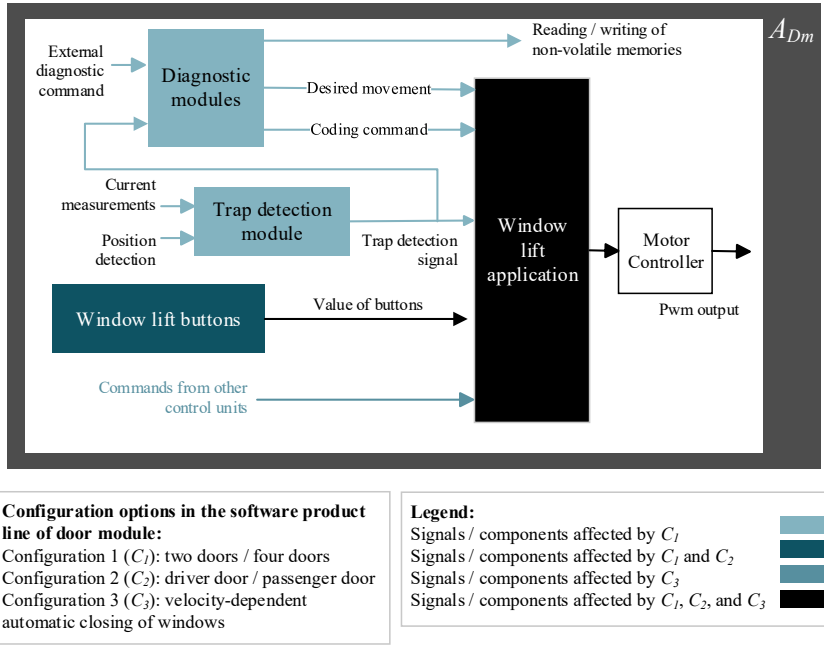


Figure 6.9: The application layer of S_{Dm} for variant-richness analysis

It is here assumed that the software system must address the variability-relevant requirements, aligning with the requirement V_1, V_2, and V_3 presented in Section A.4.2. Based on these variability requirements, three independent configurations are derived and listed as follows, see also Section A.5.2:

(C_1) Type of the car: 2-door or 4-door. As the functions for the rear doors must be realized explicitly, this configuration affects all of the shown SW-Cs in Figure 6.9.

(C_2) Position of the door module: Driver or passenger. Due to the requirement F_2 in Section A.4.2, the driver must be enabled to control all the windows, while the passengers can only control the corresponding window. Therefore, this configuration affects the input SW-C *Window lift buttons* and the subsequent application SW-C.

(C_3) Automatic closing feature: Enabled or disabled. The automatic closing relies on the vehicular velocity that is sent periodically within the signal *Commands from other control units* from the connected ECU. Therefore, this configuration influences only *Window lift application*.

Within the software system S_{Dp}, it is assumed that the control algorithm is well written and therefore the SW-C *Motor controller* is variant-independent.

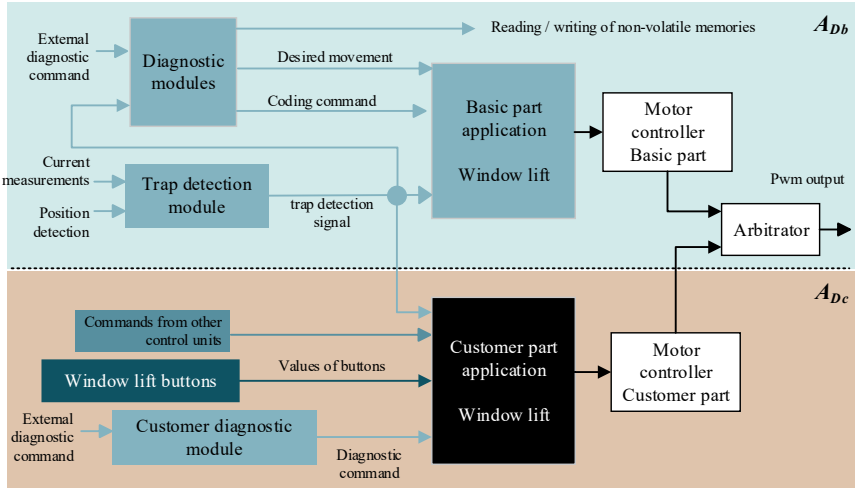


Figure 6.10: The SW-Cs at the application layer of P_{Db} and P_{Dc} for variant richness analysis, cf. Figure 6.9.

Using the concept of POSP, the software system S_{Dm} can be refactored, as illustrated in Figure 6.10. As a result of the refactoring step reducing overlap, see Section 5.3, the basic part does not include software related to button inputs. Additionally, the application software within the basic part is simplified compared with the original implementation of S_{Dm}. By removing button inputs from other

control units, the application SW-C in the basic part becomes independent of configurations C_2 and C_3. Additionally, the SW-C *Motor controller* and the arbitrator module are variant-independent.

6.5.2 Software partitioning with variability realization mechanisms

To quantify the costs associated with variant richness of software systems with variability at the SW-C level, this thesis proposes metrics based on variability realization mechanisms, detailed in Section A.5.3. The more SW-Cs are affected by variability, the higher the variant richness costs of the system become, which in turn leads to increased maintenance efforts for the software family as customer requirements evolve.

Software partitioning with the annotative variability realization mechanism

For the annotative variability realization mechanism (see Figure A.11), the metric variability realization index V_r is defined as the total number of possible implementation forms of SW-Cs within the examined software system. The variability realization index V_r of a software system S can be calculated by

Definition 6.2: Variability realization index (V_r)

$$V_r(S) = \sum_{i=1}^{j} 2^{O(c_i)} \tag{6.19}$$

where c_i represents the i-th SW-C with variants, j the number of SW-Cs with variants in the considered system, and $O(c_i)$ the number of possible configurations of the SW-C.

For the monolithic software system S_{Dm} depicted in Figure 6.9, the variability realization index is computed as:

$$V_r(S_{Dm}) = 2^1 + 2^1 + 2^2 + 2^3 = 16 \tag{6.20}$$

The variability realization index of the basic part software system P_{Db} (shown in Figure 6.10) can be calculated as:

$$V_r(P_{Db}) = 2^1 + 2^1 + 2^1 = 6 \tag{6.21}$$

As discussed above, the SW-C *Window lift buttons* shown in Figure 6.9 and Figure 6.10 is affected by C_1 and C_2. As a result, the SW-C *Window lift application* in P_{Dc} is affected by all the configurations. Accordingly, the variability realization index of the customer part P_{Dc} can be quantified as:

$$V_r(P_{Dc}) = 2^1 + 2^2 + 2^3 = 14 \tag{6.22}$$

Considering the entire partitioned software system, the total variability realization index is determined by summing the indices of the basic and customer parts:

$$V_r(P_{Db} + P_{Dc}) = V_r(P_{Db}) + V_r(P_{Dc}) = 20 \tag{6.23}$$

Although the software after the partitioning, namely $\{P_{Db}, P_{Dc}\}$ shown in Figure 6.10, is more sophisticated regarding the variability realization index, the refactoring towards POSP significantly reduces variability realization index of each partitioned software sub-system, particularly concerning production usage, see Equation 6.21. This reduction minimizes risks during end assembly by allowing more thorough testing with fewer variants.

Additionally, the variability realization index of the customer part P_{Dc} is reduced compared with the original monolithic software system S_{Dm}, as the SW-C *Trap detection module* is removed. The reduction of variability realization index and

the number of SW-Cs simplify the further software development process. Notably, the basic and customer parts are decoupled with help of the partitioning approach, enabling independent further developments.

Software partitioning with compositional variability realization mechanism

In the context of the compositional variability realization mechanism (as shown in Figure A.11-(b)), a core model serves as the foundation upon which *supplementary software modules* can be added to enable specific functionalities. For the vehicular power window system, the core model is configured as *2-door, Passenger* and *Automatic closing disabled*, as this variant corresponds minimal effort. For example, a module responsible for processing vehicle velocity can be incorporated into the *Window lift application*, enabling velocity-dependent automatic closing. Due to the usage of the compositional variability realization mechanism, it is assumed that the SW-Cs can be enhanced by *supplementary software modules* within the considered SW-C.

Given that the software system S_{Dm} (shown in Figure 6.11) should be analyzed regarding variability using the compositional variability realization mechanism, the core module is configured as follows: *Two doors, Passenger seat, and Deactivated automatic closing.*

In Figure 6.11, SW-Cs can contain smaller internal blocks. The blocks on the left represent supplementary software modules related to input signal processing, whereas those on the right correspond to modules for output signal processing.

This thesis formulates the metric variability usage index (U_i) to quantify the variant richness costs in case of the usage of the compositional variability mechanism. The metric is defined as:

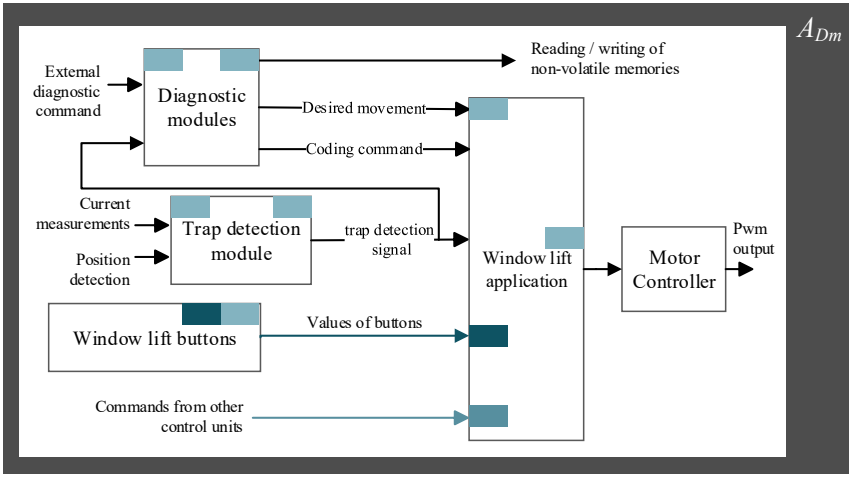


Figure 6.11: S_{Dm} with usage of compositional variability realization mechanism

Definition 6.3: Variability usage index (U_i)

$$U_i(S) = \frac{1}{v} \sum_{i=1}^{j} m_i \tag{6.24}$$

in which v is the number of possible variants, j the number of SW-Cs with variants in the considered system, and m_i the number of added software modules for the variant i.

For the monolithic software system S_{Dm}, which has 8 variants due to 3 independent configurable features with 2 options each, the variability usage index can be computed as:

$$U_i(S_{Dm}) = \frac{1}{8} \cdot (0 + 1 + 7 + 8 + 2 + 3 + 9 + 10) = 5 \tag{6.25}$$

Table 6.5 details the number of supplementary modules for each variant of S_{Dm}. The core model does not use any supplementary software module. Therefore, the corresponding m_i in Equation 6.24 for the core module is 0.

Table 6.5: Number of added software modules for each variant of the monolithic software system S_{Dm} with usage of compositional variability realization mechanism

	Automatic function deactivated	Automatic function activated
Passenger 2-door	0	1
Passenger 4-door	7	8
Driver 2-door	2	3
Driver 4-door	9	10

Following the same principle, for the partitioned software system shown in Figure 6.12, the variability usage indexes are calculated for the basic part (P_{Db}) and the customer part (P_{Dc}) as:

$$\begin{aligned} U_i(P_{Db}) &= \frac{1}{2} \cdot (0 + 6) &&= 3 \\ U_i(P_{Dc}) &= \frac{1}{8} \cdot (0 + 1 + 5 + 6 + 2 + 3 + 7 + 8) &&= 4 \end{aligned} \tag{6.26}$$

From the comparison between the variability usage index of the basic part P_{Db} and the monolithic software system S_{Dm}, partitioned software reduces the usage of supplementary software modules for each software part derived from refactoring of POSP. Additionally, the reduction in the usage index from the monolithic system

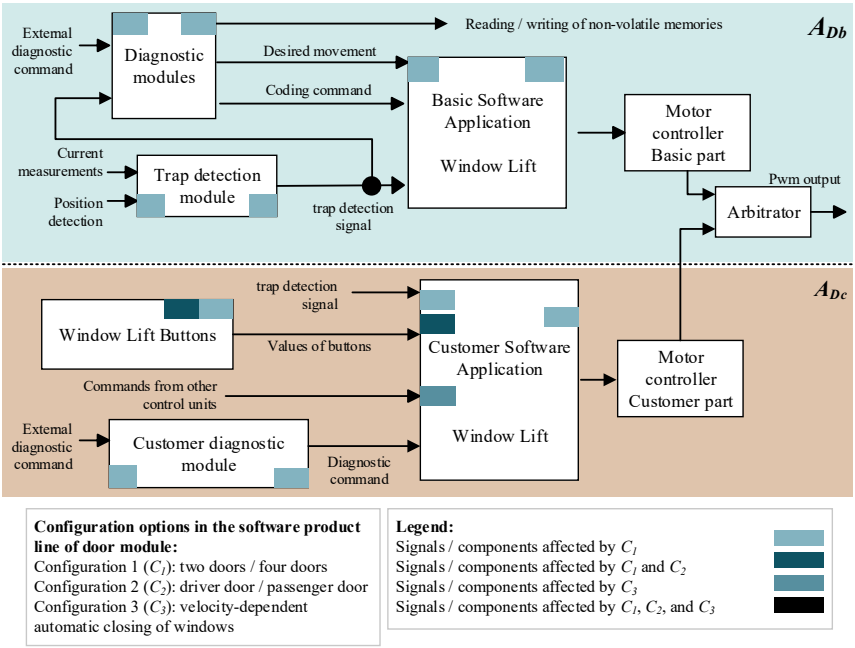


Figure 6.12: P_{Db} and P_{Dc} with usage of compositional variability realization mechanism

to the customer part software indicates simplified software development in terms of variability.

6.5.3 Evaluation of variant richness

Assuming the cost of variant richness of software systems considered can be reflected by the presented metrics variability realization index (Equation 6.19) and variability usage index (Equation 6.24), S_{Dm} and S_{Dp} are compared using these metrics. Higher values of these indices indicate a higher cost associated with variant richness.

For both variability realization mechanisms under study, the cost for P_{Db} of variant richness is lower than that of S_{Dm}, as

$$V_r(P_{Db}) < V_r(S_{Dm}) \tag{6.27}$$

At the same time, the comparison indicates that software refactoring increases the cost of variant richness:

$$\begin{aligned} V_r(S_{Dp}) &> V_r(S_{Dm}) \\ U_i(S_{Dp}) &> U_i(S_{Dm}) \end{aligned} \tag{6.28}$$

More generally,

- Partitioning a monolithic software system increases the overall variant richness of the considered system;
- Each derived software part has lower variant richness than the original system.

6.6 Discussion

The implementation of the software system pair $\mathscr{S}_D = \{S_{Dm}, S_{Dp}\}$ serves as proof-of-concept for POSP, demonstrating that physically partitioning µC-based automotive software is feasible not only in theory but also in practice, as evidenced by the following:

- S_{Dm} is developed based on MICROSAR that implements production-grade CP.
- S_{Dp} is derived based on S_{Dm} with usage of SwCluC with full compatibility to MICROSAR.

- The development and realization of S_{Dm} and S_{Dp} follows series-development tool chains, such as DaVinci Developer, DaVinci Configurator, and CANdelaStudio [162].
- The µC, on which S_{Dm} and S_{Dp} are implemented, is the same for series production ECUs.

Therefore, $\mathscr{S}_D$ ensures that the prototype reflects real development conditions, proving the viability of POSP under industry-grade constraints.

The sections above evaluated the derived partitioned software system S_{Dp} quantitatively using various metrics. In summary, the quantitative evaluation highlights the following findings:

- Refactoring automotive software systems utilizing POSP can achieve a notable reduction in software size for in-line flashing purposes. In the developed prototype, the effect was demonstrated by a production reduction ratio δ_{pr} of 16%, as derived in Equation 6.11. Due to the small scale of the implemented prototype software, the proposed concept is expected to be even more effective as the size of the considered system increases.
- The partitioning towards the basic and customer parts leads to increased demand for memory due to partial duplication of SW-Cs during the refactoring and the inclusion of additional BSW modules in CP. The newly provoked memory usage may not be compensated by the benefit of software size reduction.
- The refactoring separates the structural responsibilities of the software system considered effectively, leaving the structural load at the SW-C level within the basic part and steering the complication at the Runnable level into customer parts.

For the raised issues presented in Chapter 1, it is important to emphasize that a software part (Definition 4.1) that is oriented to production usage (the basic part)

is able to be derived for legacy software systems both theoretically and practically so that:

- The commissioning and testing procedures can be fully dependent on the basic part, regardless of presented usage scenarios (Figure 4.11). The basic part remains stable and independent of software updates. This addresses issue I_1 given in Chapter 1 (see Page 4).
- Production-relevant applications can be developed and integrated independently of software updates. These applications can be built within application SW-Cs in the basic part. This statement corresponds to issue I_2.
- Only a subset of software (either the basic or the customer parts, depending on scenarios shown in Figure 4.11) must be installed in the end assembly line, which minimizes the in-line flashing duration and resolving issue I_3.

7 Summary and Outlook

7.1 Summary

In-car software is fundamentally reshaping the automotive landscape. For car owners, the expansion of software capabilities enables frequently evolving features, connected and automated driving, and a more personalized user experience that extend beyond the vehicle itself [129]. However, the amount of technical challenges that OEMs face today are even higher due to the more significant role of software for automotive innovations [43]. First, the software innovation requests for corresponding adjustment of hardware platforms, leading to the transformation of automotive E/E systems and E/E architectures as outlined in Section 2.3. Second, the software-driven vehicle functionalities require in-car software to be easily updatable after initial deployment [61]. Third, the adoption of automotive standards results in a growing number of safety and security features that must be addressed in automotive software systems [122].

Meanwhile, the needs of manufacturing are often overlooked during the wave of software expansion in the automotive industry, as the manufacturing encompasses diverse activities (discussed in Section 2.1), and the intersection between software developers and production processes remains limited. Nevertheless, frequently evolving in-car software often induces side effects to the efficiency and stability of automotive production, especially during the end assembly, as stated in Section 3.5. Especially, the growing size of in-car software influences the duration of in-line flashing, impacting the efficiency of the end assembly. Furthermore, frequent and regular software updates can trigger commissioning and testing errors that incur costly end-of-line manual rework or even an unplanned production stop.

These phenomena represent significant financial losses for OEMs, as detailed in Section A.1.

Facing these software-induced challenges, various approaches have been attempted to tackle the software-related difficulties in the end assembly without altering the underlying software architecture, as presented in Section 3.1. However, these methods are often expensive, demanding, or restricted, proven to be ineffective against production issues stemming from in-car software. Therefore, in-car software architecture must be tailored to ensure the efficiency and stability of the end assembly of automotive manufacturing.

Against the background, this thesis proposes and answers the following research questions (see also Section 1.2) as its scientific contributions:

> RQ_1: *What are the key characteristics of a production-optimized automotive software architecture that ensures efficiency and stability of the end assembly?*

The idea of a production-optimized automotive software architecture is the separation of the concerns between production and software development. Following this idea, the thesis proposes partitioning the in-car software into the basic and customer parts, where the basic part is responsible for production-relevant activities and customer parts are driven by the latest requirements. These parts must be separated physically and compiled individually to ensure usability, as stated in Section 4.1. Such physically separated software systems are termed POSP, corresponding to a reversed L-form distribution in a layered architecture.

Several design characteristics are derived from the concept of POSP as detailed in Section 4.3. First, inter-part communication interfaces must be established to enable information exchange between the basic and customer parts. Second, arbitration mechanisms must be designed to handle conflicting output signals from different parts. Third, diagnostic interfaces are preferred to be implemented within the basic part. Fourth, usage of POSP leads to additional resource usage.

> RQ_2: *How can the existing AUTOSAR Classic-based automotive software systems be refactored to align with the proposed software architecture?*

To illustrate an approach that refactors legacy software systems according to POSP, this thesis proposes a generic simplified model describing automotive software systems at the application layer with usage of the AUTOSAR Classic Platform as discussed in Section 5.1.

Based on the generic simplified model, a workflow is outlined to transfer µC-based software systems to align with the proposed POSP, as detailed in Section 5.2. The workflow considers the criteria in case of POSP usage. Additionally, the workflow takes deployment, verification, and evaluation of the software refactoring into consideration. Several metrics are proposed to identify the characteristics of considered software systems and to evaluate the result of software partitioning.

The core part of the workflow is a partitioning approach that partially duplicates the considered SW-Cs. The resulting redundancy-based partitioning approach is presented in Section 5.3. During the processes of the proposed partitioning approach, realization of diagnostic interfaces is released, followed by duplicating some of the SW-Cs in the system, which enables to branch input signals and integrating arbitrators. Then all the SW-Cs can be assigned to newly created basic or customer parts with the consideration that inter-part communication interfaces must be configured properly.

The proposed concept and methodology are validated through a prototype implemented on an industry-grade microcontroller, as illustrated in Chapter 6. Developed based on series-production procedures and tool chains, the prototype utilizes the AUTOSAR Classic Platform and the mechanism Software Cluster Connection (SwCluC), demonstrating the feasibility of POSP and its potential for short-term integration in series-production systems.

> RQ_3: *What implications arise from the usage of the proposed software architectural changes within the AUTOSAR Classic software lifecycle, particularly for automotive production?*

Refactoring a software system using the proposed approach results in one basic part and several customer parts, who can be installed separately.

The physically separated basic and customer parts allow for a changed deployment strategy within the end assembly, as discussed in Section 4.4.2. Especially, only partial software must be installed during the end assembly. Additionally, commissioning and testing rely only on the basic part, which brings the following benefits for production:

- Installation of partial software reduces the software size to be flashed (analyzed in Section 5.2 and practically evaluated in Section 6.3), hence reducing the in-line flashing duration directly.
- Commissioning and testing results only depending on the basic part stabilizes the results of these procedures, improving the stability of production and reducing the possibility of manual workarounds and unplanned production stops.

However, refactoring legacy software systems incurs corresponding costs:

- Refactoring demands analysis of software systems at the signal level and manual verification, as stated in Algorithm 1, requires development efforts.
- Refactoring increases resource demands, including memory and computing power, potentially necessitating µC upgrades.

Architecturally, partitioning software decomposes the structural responsibilities, leaving the structural load of at the SW-C level within the basic part and steering the complication at the Runnable level into customer parts, as shown by the prototype (see Section 6.3.2).

In summary, implementing POSP can effectively raise the efficiency and stability of software usage in the end assembly. However, these benefits come with trade-offs in development effort and potential hardware resource requirements, which must be evaluated for each considered ECU.

7.2 Outlook

The research presented in this thesis demonstrates that POSP offers a viable and effective architectural approach to address the side effects of increasingly comprehensive and volatile in-car software on the efficiency and stability of automotive end assembly.

While this thesis establishes the fundamental concepts, methodology, and evidence for a production-optimized automotive software architecture, several promising directions remain for future research:

1. *Extension to service-oriented platforms*: This thesis focuses on the application of POSP in AUTOSAR Classic Platform, a signal-based architectural platform. A natural step is to extend the approach on service-based platforms, such as AUTOSAR Adaptive Platform (AP) (Section 3.2.2). Although AP-based ECUs currently represent a minority in vehicles, their occurrence and functionalities are increasing and must be considered regarding in-line flashing and commissioning in the end assembly.

2. *Automation of software refactoring*: The proposed partitioning approach (Section 5.3) assumes manual refactoring by developers, which incurs additional development costs and potential for errors. However, since automotive software is typically documented in machine-readable formats (e.g., AUTOSAR XML (.arxml) files), artificial intelligence methods could be employed to automate labor-intensive tasks, such as detailed signal-level analysis and the assignment of SW-Cs according to given requirements.

3. *Tool chain improvements*: The developed prototype of the thesis uses the mechanism SwCluC (Section A.6.2) based on MICROSAR. Compared to traditional monolithic software on CP, this partitioned implementation is much more demanding. For example, four projects instead of a single project must be maintained to ensure the correct generation of BSW and RTE. Therefore, it is desired to simplify and improve the tool chain for partitioned software development of CP. Additionally, the binary files of the basic and

customer parts generated by compilers must be specially handled to ensure the consistency of binary manifests (Figure A.13). This special processing must be optimized out to ensure a flexible update of customer parts.

4. *Outsourcing customer parts from ECUs*: The concept POSP paves the way to partition software features in µC-based systems within a single ECU. A more radical evolution is to outsource customer parts entirely from the ECU to upper machines (such as domain-centralized controller [Z6]). This shift does not demand the usage of physical software partitioning mechanisms (e.g., SwCluC). By moving features out of the resource-constrained ECU, updates can be performed with more flexibility and without modification of the embedded system, making ECUs more robust. However, outsourcing features from µC-based systems increases bus communication load, as more signals must be transmitted with even higher frequency. Furthermore, it is unsuitable for safety-critical features with low-latency requirements, as the delay of bus communication can violate the real-time requirement.

Beyond these technical refinements, two broader research perspectives also emerge:

1. *Tailored development process*: The thesis has examined the refactoring of legacy software systems according to POSP. An open question remains: how can partitioned systems be configured and developed from scratch? A promising idea is to decouple the development of the basic and customer parts to accelerate the path from development to series production. However, such a decomposition requires careful consideration of hardware design as well as safety and security concerns.

2. *Software partitioning for software-hardware decoupling*: Organizing software according to POSP is only one perspective on partitioning. In addition to the target for production efficiency and stability, partitioning could also facilitate decoupling between software and hardware in ECUs. One potential application is partial software updating in response to hardware changes, which would require sensor and actuator drivers to be implemented in dedicated software parts (e.g., a customer part).

A Appendix

A.1 Economic consequences of production instability and inefficiency

Architectural redesign and rethinking are time-consuming and expensive, particularly in the context of automotive software. In-car software must meet stringent criteria across multiple dimensions, and its development typically involves numerous stakeholders throughout the value-added chain. Consequently, tailoring software to align with production needs requires strong evidence. This section provides an estimate of the economic consequences that production instability and inefficiency that are potentially caused by in-car software.

A.1.1 Cost of unplanned downtime

The most extreme case that software issues can have on the manufacturing process is unplanned stop of the final assembly line. Unplanned downtime is costing ever more than in the history. According to recent reports, the cost of an hour of downtime in an automotive plant has risen to over $2 million, compared to $1.3 million in 2019–2020 [140].

Using the August 2024 exchange rate ($1 = €0.91), a single minute of unplanned production downtime results in a financial loss of approximately €30,000 for the OEM.

Figure A.1: Vehicle movers used in the automotive end assembly, based on Stringo AB [148]

At the end of the assembly line, the engine of the produced car must be started immediately after the commissioning and Electrical and/or Electronic (E/E) tests, see Section 2.1.2 and Figure 2.3. Unsuccessful commissioning or incomplete software flashing at this stage can directly impact the drivability of the vehicle, necessitating special transportation and manual rework in parking slots outside the production halls. Therefore, unplanned production stops are often caused by software issues in the following two scenarios:

- *Congestion at the end of production line:* Continuous failure of in-line flashing or commissioning (see Figure 2.3) end can cause congestion at the end of production. These undrivable cars must be transported using vehicle movers (see Figure A.1) to keep the assembly line moving. If these failures continue and there are insufficient staff or vehicle movers to handle the backlog, the assembly line must stop and wait.
- *Overcapacity of the reserved parking slot:* Transporting undrivable cars away from the end of the production line is only part of the solution. These vehicles are stored on parking slots outside the production line while awaiting rework. If these parking slots reach full capacity, the production line must halt. Typically, reworking a vehicle's Electrical and/or Electronic (E/E) systems requires at least 20 minutes and the use of an external diagnostic tool (see also Figure 2.13).

A.1.2 Cost of flashing, commissioning, and testing on the assembly line

One of the main objectives of this doctor thesis is to enhance the efficiency of E/E processes within the end assembly, focusing on the in-line flashing, commissioning, and testing procedure, see also Section 2.1.2. The efficiency is directly linked to the duration, which is further linearly related to the length of the assembly line required for these procedures.

The cost associated with these processes can be estimated from the following categories:

- *Depreciation cost:* The assembly line has a limited useful life. It is assumed that the automotive end assembly line has a life cycle of 10 years, operates 250 days in a year, and runs 16 hours in a day. Additionally, constructing one meter of the assembly line is estimated to be €15000 [128].

  Based on these assumptions, the depreciation cost per meter of the production line in one minute, denoted as C_d, can be calculated as:

$$C_d = \frac{15000}{10 \times 250 \times 16 \times 60} = 0.00625€/m \tag{A.1}$$

- *Operation cost:* Operating the assembly line to transport the car body consumes energy. It is assumed that the motor of assembly lines has a power consumption of $1.5kW/m$ and the electricity cost is $0.3€/(kW \cdot h)$. The operation cost C_o can be calculated as:

$$C_o = \frac{1.5 \times 0.3}{60} = 0.0075€/m \tag{A.2}$$

- *Labor cost:* The commissioning and testing procedure demands participation of assembly staff. The labor cost is assumed to be $50€/h$ per worker [146]. Therefore, the labor cost for commissioning and testing C_{lc} is

$$C_{lc} = 0.83€/min \tag{A.3}$$

  Additionally, the in-line flashing process requires supervision by staff. Assuming one employee is needed per $80m$ of the assembly line, the labor cost for in-line flashing C_{lf} is derived as

$$C_{lf} = \frac{50}{60 \times 80} = 0.0104€/m \tag{A.4}$$

- *Maintenance cost:* Furthermore, the operation of the assembly line needs maintenance. It is assumed that 7% of the purchase price is allocated for maintenance. The maintenance cost is derived as:

$$C_m = \frac{15000 \times 7\%}{250 \times 16 \times 60} = 0.00438€/m \tag{A.5}$$

Further it is assumed that the assembly line moves at around $0.15m/s$, or $v_a = 9m/min$. The estimated cost associated with the E/E-relevant processes every produced vehicle are summarized as follows:

$$C_C = C_{lc} + (C_d + C_o + C_m) \cdot v_a = 0.996€/\text{car} \tag{A.6}$$

$$C_F = (C_d + C_o + C_{lf} + C_m) \cdot v_a = 0.257€/\text{car} \tag{A.7}$$

in which C_C corresponds to the cost of commissioning and testing per minute and C_F represents the cost of in-line flashing per minute.

A.1.3 Cost of manual rework

As discussed in Section A.1.1, the undrivable cars stored on the parking slot require manual repair. In case of software related errors, two primary actions are usually taken to restore these undrivable cars, see also Section 3.5.1.

Incomplete flashing process If the in-line flashing process is incomplete at the beginning position of commissioning and testing (see Figure 2.3), the flashing procedure will be stopped manually. Although these vehicles remain drivable, they require rework, which involves re-flashing the affected ECUs and performing the necessary commissioning and testing tasks for these newly flashed components.

While it is a common practice to re-flash all ECUs in the EE topology, it is assumed that only the specific ECUs requiring are re-flashed, commissioned, and tested. This process takes 15 minutes per vehicle. Therefore, the cost of manual rework due to incomplete flashing process per car, denoted as C_{rf} is calculated as:

$$C_{rf} = \frac{50 \times 15}{60} = 12.5\text{€/car} \tag{A.8}$$

Failed tests due to updated software If an updated version of ECU software leads to failed commissioning or testing, the vehicle must undergo rework. This rework includes re-flashing the affected ECU and re-performing the associated commissioning and testing procedures. The cost of this rework depends on the number of ECUs involved. It is assumed that the testing equipment is adjusted for the specific issue, as the software update affects all cars produced with the same configuration. Re-flashing and re-executing commissioning tasks are estimated to take 10 minutes per vehicle. Therefore, the cost of manual rework due to failed tests from updated software, C_{rc}, is:

$$C_{rc} = \frac{50 \times 10}{60} = 8.33\text{€/car} \tag{A.9}$$

Additionally, it is assumed that the production line runs with a takt time of 1 minute [85] and the assembly line runs 16 hours a day. The daily financial loss for an unsolved software update error for one assembly line can be accumulated as:

$$C_{rc_d} = 60 \cdot 16 \cdot C_{rc} = 8000€/\text{day} \tag{A.10}$$

A.2 Introduction and analysis of the AUTOSAR Classic platform

AUTOSAR Classic Platform (CP) contains three layers upon the embedded hardware, as shown in Figure A.2:

- **Basic Software (BSW)**: The BSW is hardware-related and serves the purpose of standardizing inputs, outputs, memory, and library functions of the utilized µC [22]. Moreover, BSW is responsible to realize the functionalities that enable the software components of the layers upon to be developed independently from the hardware settings. These functionalities include task management, management of non-volatile memory, diagnostic interfaces and ECU state management. According to the AUTOSAR methodology, the BSW should be configured rather than programmed directly through code [23].
- **Run Time Environment (RTE)**: Derived from the BSW, the RTE is the layer providing inter- and intra-ECU communication interfaces to the application software [22]. Moreover, the RTE triggers the tasks of Software Components (SW-Cs) in the Application Layer. Additionally, the RTE abstracts the operating system, preventing the SW-C from accessing the BSW directly. Therefore, the RTE is seen as glue, adhering to the fully configured BSW and the application software.

- **Application Layer**: The Application Layer is the layer realizing the desired functionalities of the control unit. This layer is made up of different Software Components (SW-Cs) that are called during runtime.

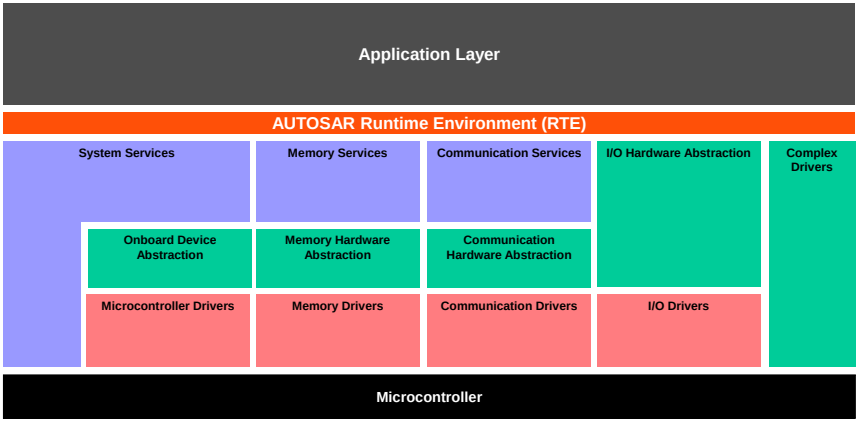


Figure A.2: Overview of software layers in the AUTOSAR Classic Platform (CP), based on AUTOSAR consortium [22].

To address the challenge of software-hardware decoupling, the CP employs the concept of Virtual Functional Bus (VFB) that separates the design of inter-ECU functions and their implementation on dedicated ECUs. Figure A.3 illustrates the application of VFB concept using the hazard light function as an example [88]. The hazard light functionality involves the driver pressing a button, causing the hazard lights on the front and rear sides of the vehicle to blink.

In the VFB perspective, SW-Cs are designed without being tied to specific ECUs, thus abstracting away considerations of inter-ECU communication. Taking the hazard light function as an example, the SW-C *HazardLightApplication* is responsible for implementing the control logic. It receives input signals from *HazardLightButton* and transmits outputs to *LedHazardLightFront* and *LedHazardLightRear*.

Moreover, the SW-C *HazardLightApplication* utilizes the BSW module ECU State Manager to obtain information about the state of the µC. In the context of actuator SW-Cs, the Digital Input/Output module within the BSW is leveraged.

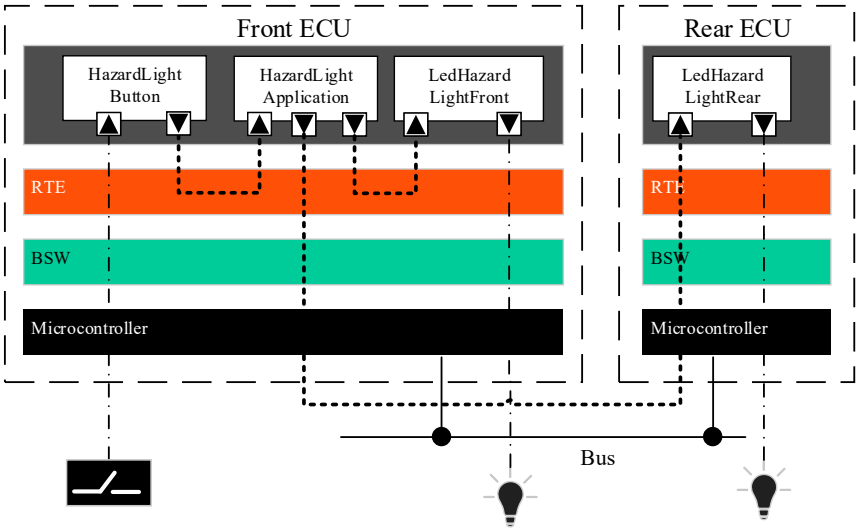


Figure A.3: Implementation of the hazard light function into ECUs.

Once the functions in VFB perspective are finalized, the software architecture design for the determined ECU can be derived. Assuming the LEDs for hazard light are controlled by two separate ECUs, *Front ECU* and *Rear ECU*, the hazard light function necessitates inter-ECU communication. Figure A.3 showcases the implementation of the hazard light function, where the BSW realizes the communication channel between ECUs. The RTE connects the interfaces of SW-Cs to the corresponding communication channel.

For instance, the communication between the button and the front LEDs remains intra-ECU, meaning it occurs within the RTE of *Front ECU*. Conversely, the control signal for rear LEDs relies on CAN buses. As a result, the RTE directs the output port of *HazardLightApplication* to the BSW. The periodic transmission of signals on the CAN bus is then managed and executed by BSW.

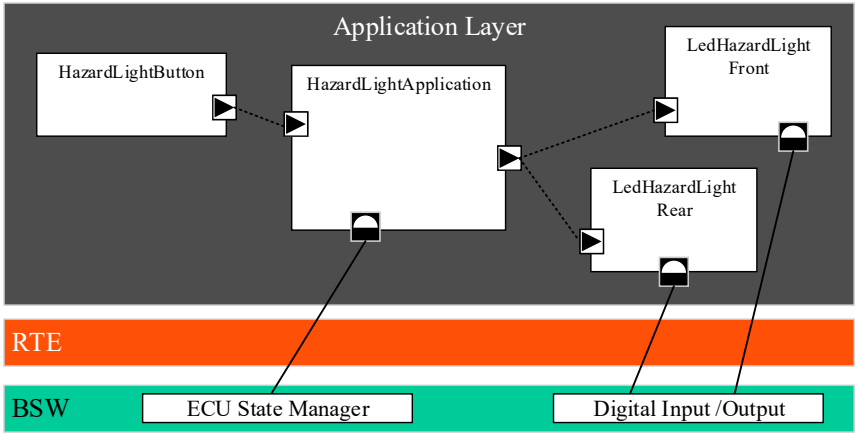


Figure A.4: Example software design of the hazard light function in the view of AUTOSAR Virtual Functional Bus (VFB), based on Kindel et al. [88]. The icons follow the Sender/Receiver interface of AUTOSAR [20].

As the example of hazard light function shows, the usage of the AUTOSAR Classic Platform (CP) helps to achieve a crucial goal of software development: separation of concern. This separation allows OEMs to focus on application development, specifically on the SW-C design within the VFB view, without delving into the hardware-related issues, as demonstrated in Figure A.4. The framework of the CP ensures that the hardware realization is abstracted, enabling interchangeability of components from various suppliers.

Additionally, the CP mandates semiconductor manufacturers to deliver standardized modules for BSW, which enables the interchangeability of µCs. Moreover, the regulated development methodology contributes to the reusability of both software and hardware designs. As a result, the usage of the CP elevates the product quality and reduces the development cost for microcontroller-based systems in the automotive industry [11].

However, the benefits of standardization in the CP come with associated challenges. The of the platform is evident from the documentation, numbering over 200 documents [24]. Complicating matters further, the AUTOSAR consortium does

not provide the implementation of the platform itself. Instead, system suppliers, such as Vector Informatik's MICROSAR [160] and Elektrobit's EB tresos [49], handle the realization of the AUTOSAR platform. The reliance on commercial tool chains leads to an initial investment for developers.

The challenges continue when it comes to generating operational binary files for µCs. The seamless integration of all tool chains and configurations is demanded, making common practices like coding, debugging, and testing challenging for a single engineer. The high license costs and steep learning curve create barriers, especially for newcomers attempting to familiarize themselves with the Classic AUTOSAR ecosystem [57].

Looking ahead, the usage of the CP is restricted due to the stringent and inflexible methodology, particularly regarding frequent software updates. The communication matrix precedes the software architectural design. Consequently, introducing a new signal into the communication matrix leads to a new deployment and compilation of the µC, and potentially even the connected ECUs. This lack of flexibility prompts rethinking of the CP.

A.3 Quality metrics for automotive software systems

As illustrated in Section 3.3.4, the software components used in the automotive industry are not developed in-house at OEMs, but outsourced to Tier 1s and suppliers. Therefore, software systems have to be evaluated to meet quality standards of OEMs.

The standards and norms, for instance, ISO/IEC 25010 [80] provides overviews of software quality criteria. For instance, Figure A.5 illustrates software product quality attributes according to ISO/IEC 25010. However, these criteria are often not quantified for the software evaluation with metrics [166].

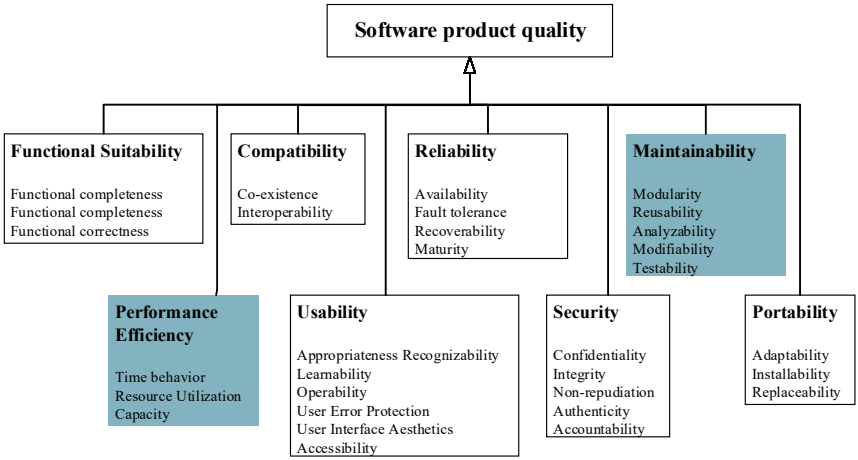


Figure A.5: Overview of software product quality attributes, based on ISO/IEC25010 standard [80].

In theory, three categories are given to evaluate the software quality by utilizing metrics [166]:

- **Process metrics**: quantifying properties within software development processes,

- **Product metrics**: quantifying properties of products represented by models and architectural documentation, and
- **Code metrics**: quantifying properties regarding source codes.

ISO/IEC 25010 introduces 8 sub-categories to evaluate the software product quality. Among these, the thesis places emphasis on the topics of maintainability and performance efficiency as depicted in Figure A.5. While the criterion maintainability is concerned with production stability regarding frequent software updates, the performance efficiency is related to the efficiency of in-line flashing.

In the example development process outlined in Section 3.3.4, architectural design and source code development are distinctly separated. SW-Cs are designed independently of the source code, with the actual code being developed based on generated code templates derived from these SW-Cs.

As a result, this section introduces relevant software metrics that are used to evaluate concepts in following chapters and proposes potential metrics with regard to software usage in the end assembly process as follows:

1. **Lines of Code (LC)**: This metric is calculated by the number of executable source instruction excluding comments and blank lines, indicating the complexity of the software system / software components [174].

$$LC = \#CodeLines - \#Comments - \#Blanks \tag{A.11}$$

2. **McCabe Cyclomatic complexity number** (M_{cc}): This metric measures the amount of independent decision logic through a program, representing the testing complexity [70]. In a control flow graph, assuming each node corresponds to a block of sequential code, and each edge corresponds to a path created by a decision.

   M_{cc} can be calculated by

$$M_{cc} = e - n + 2p \tag{A.12}$$

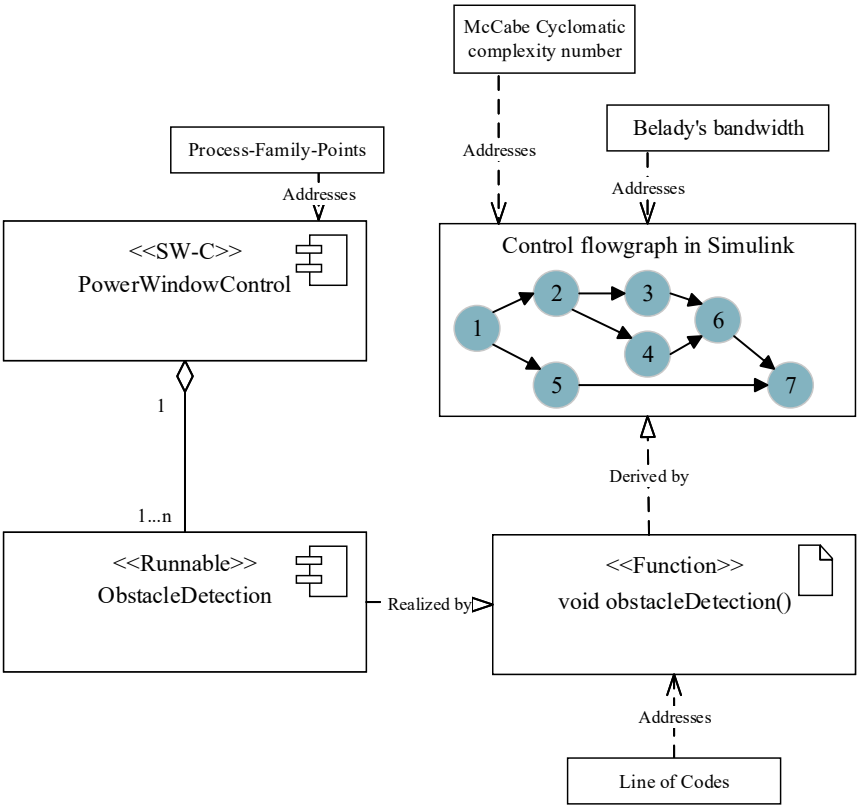


Figure A.6: Possible usages of selected metrics within the AUTOSAR Classic Platform (CP), illustration of control flowgraph based on Huda et al. [70].

in which e is the number of edges, n is the number of nodes, and p is the number of connected module components.

3. **Belady's bandwidth metric (BW)**: This metric indicates the average level of nesting or width of the measured control flow graph [174].

$$BW = \frac{1}{n} \sum_{i} iL_i \tag{A.13}$$

4. **Process-Family-Points (PFP)**: This metric indicates structured reuse of components degree of automation based on a common infrastructure [87].

Figure A.6 illustrates possible usages of these metrics within the CP-based software development. An SW-C can contain several Runnables which is realized by functions in programming language C. The function can be derived by model-driven tools, such as Simulink. The logic design in Simulink can be seen as control flow graph. While the metric Process-Family-Points addresses the reusability on the level of SW-C, the other metrics concentrate on the level of source codes and control flow graph.

Against this background, despite the huge number (112 reviewed research papers in 2020 [166]), the introduced metrics in literature for software quality are generic, and not specifically designed for automotive systems. Additionally, an efficient and flexible measurement tool for quantifying the metrics, is missing [166]. Regarding the usage of automotive software within the manufacturing process, especially the end assembly process, there is little metrics developed to evaluate automotive software systems.

In context of software usage in the end assembly, metrics should be developed regarding key procedures, namely in-line flashing, commissioning, and testing, with respect to diagnostic-based processes (see also Section 2.4.3). Therefore, software metrics can be formulated to address these topics in the following manners:

- **Range of diagnostics**: Assuming a software system or component is diagnostic-relevant and supposed to be used within the end assembly, see also Section 2.4.3, metrics should be developed to quantify the dependency range of diagnostics. This could include metrics such as the number of variables that can be read/written by diagnostic commands and the range of actions that can be triggered by diagnostic commands. These metrics can indicate the relevance of commissioning and testing after software release.
- **Intersection of functionalities for multiple stakeholders**: In a software system or subsystem, SW-Cs can be required by different stakeholders. For example, the control of ambient lighting LEDs is required by production

usage, while SW-Cs related to light strips are not relevant for production needs. Therefore, metrics should be developed to quantify the overlap within software systems or subsystems that are required for production-related usages and those that are not.

A.4 Case study: vehicular power window system

The thesis aims to apply its concept (Chapter 4) and methodology (Chapter 5) to vehicular control units, using the vehicular power window system as a basis for software system analysis. The power window was chosen as an example for the following reasons:

- *Ease of understanding:* The operation of a power window is straightforward and widely recognized. Most modern vehicles are equipped with electrically operated windows, which are moved upwards or downwards by a electric motor.
- *Relevance in Production:* The vehicular power window is involved in the end assembly process. Especially, the minimum and maximum positions of the window glass are calibrated, and the functionality of the motor is tested.
- *Configurability and Extensibility:* While the electronic hardware for power windows is modular, the software functionality can differ based on the location of ECU (e.g., passenger side vs. driver side) and the vehicle type (e.g., two-door vs. four-door). Moreover, the features can be expanded by considering signals such as vehicle speed, or voice commands from the head unit.

A.4.1 Overview and rationale

The vehicular power windows enable the convenient raising and lowering of the vehicle's side windows. The movement of the side windows can be controlled by switches. As long as a switch is pressed, the respective window moves in the desired direction. If the switch is fully pressed, the window moves automatically to the lower or upper end position [69].

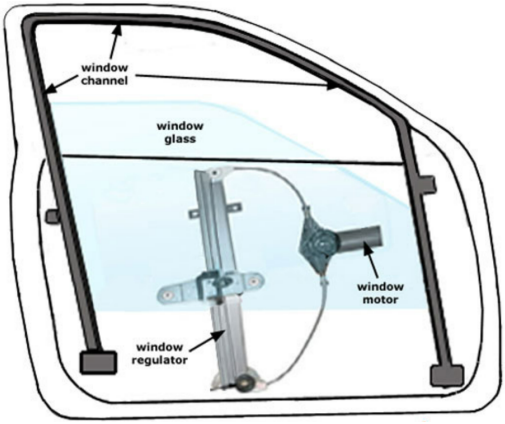


Figure A.7: The power window actuators in the car, based on Altazan [4].

The primary functionality of vehicular power windows is centrally implemented within the door module, also referred to as the door control unit in some literature. The corresponding actuators for power window functions are shown in Figure A.7. This thesis adopts the design principle that a vehicle has separate door modules for the driver and passenger doors, each controlling the windows of its respective door, as well as the corresponding rear doors in the case of four-door vehicles [69]. In modern vehicles, however, each door is equipped with its own control unit.

The driver's door is equipped with buttons dedicated to the control all available power windows. For example, these buttons are responsible for the windows of the driver's door itself, the passenger door, and the rear left and right doors in cars with four doors. Conversely, other doors are equipped with buttons that specifically control their respective windows. Additionally, a child protection switch on the driver's door can deactivate the functionalities of the power window buttons on the rear doors, providing an added layer of safety and control.

Feature description

In addition to the basic functionality where buttons trigger the movement of the corresponding window, a feature of power windows is the trap detection function. This mechanism serves as the foundation for *anti-trap protection*, a safety measure that continuously monitors obstacles during the window closing process. If an obstacle is detected, the closing sequence is immediately halted, and the window is automatically reversed to a fully open position.

Moreover, the window movement will not be executed and an error message is sent via bus communication if the battery voltage at the start of the window movement is below the given threshold.

The tasks regarding commissioning and testing should be realized by the diagnostic communication via CAN bus. In principle, three types of communication should be given.

- *Reading* is used to read out the desired value of a given variable or from a certain address from ROM or RAM of the ECU,
- *Info* (Information) is used to read out and track the desired value continuously, and
- *Writing* is used to overwrite the value of a given variable or from a certain address from RAM.

A.4.2 Derivation of technical requirements

Based on the introduction in the last paragraphs, requirements on the prototype power window systems are derived in the following categories:

Customer feature requirements

(F_1) The power window software system must drive the corresponding motor according to the button inputs.

(F_2) The driver's seat must have controls for all the car windows.

(F_3) In case that two button inputs have different demands, the motor cannot move.

(F_4) If a button is pressed over 2 seconds, the window should be driven to the end position.

Safety requirements

(S_1) If an obstacle (trap) is detected between the window and the frame during the closing process, the window must immediately halt and be driven to a fully open position for safety.

(S_2) The anti-trap protection must have higher priority than the button inputs.

Diagnostic-relevant requirements

(D_1) The power window application must provide CAN communication interface to enable the diagnostics from the external testing device.

(D_2) The diagnostic interface must be given to write the type and the series number of the control unit to non-volatile memory.

(D_3) The diagnostic interface must be given to check whether an obstacle is detected.

(D_4) The diagnostic interface must be given to deactivate the trap-protection function

(D_5) Diagnostic interfaces must be given to drive window motors.

Variability-relevant requirements

Modern software in the automotive industry comes in numerous variations, each tailored to address specific customer needs by incorporating slight variations in functionality [138]. The door module application can vary according to the position and the car type. Moreover, some advanced functions should be activated or deactivated according to customer configuration.

(V_1) The power window application must accommodate both two-door (Cabriolet) and four-door (Limousine) car configurations.

(V_2) The power window application must accommodate the door module of driver and passenger seat.

(V_3) The power window must enable the choice, whether the velocity-dependent automatic closing of the power window is activated.

A.5 Software reuse in automotive industry

The automotive industry is facing challenges from the evolving software [38]. Especially, the frequently changing software brings risk to the manufacturing process regarding flashing, commissioning and testing procedures on the end assembly line, as discussed in Section 3.5.1. Moreover, the evolving software leads to exponentially increasing numbers of software versions and variants [138]. Therefore, the influences of the variety of automotive systems on the production should be explored.

Until today, the variability management of automotive systems is based on the hardware [141]. In other words, an ECU is the unit to define a variant. If the software for an ECU is updated, a new variant should be named after the combination of the ECU hardware and the updated software. This principle only works properly if the vehicular software is never or hardly updated after the start of production. However, it will be a common practice to update the software on purchased cars to fix bugs and enhance functionalities. Therefore, new methodologies are desired to manage the variants caused by frequent software updates.

To discuss the interaction of software partitioning and variant management in the automotive industry, this section presents the basis of software reuse and variant management in the automotive industry.

A.5.1 Principles of variant management in automotive industry

In the automotive industry, it has been established that variability management is carried out at the level of control units. Due to the complexity of the vehicle system, responsibilities for subsystems are often outsourced by ECU [42].

The vehicles of Mercedes-Benz AG can be ordered in various model series. In addition to model series, the ordered car can be configured with different engines,

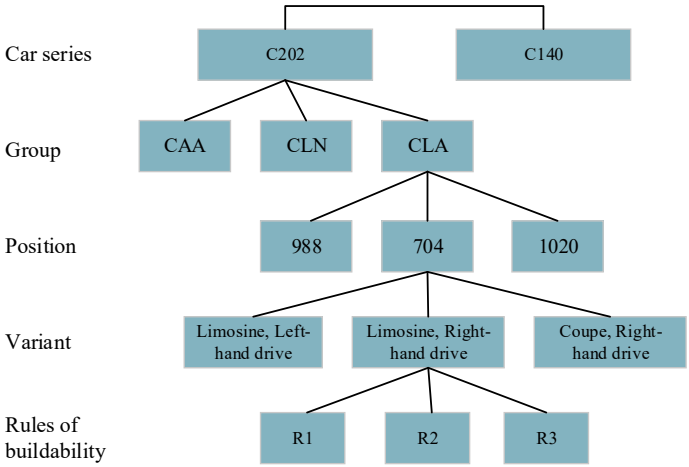


Figure A.8: Usage of build-ability check for vehicular variants, based on Sinz [141].

transmission options, and a vast assortment of special accessories, colors, and interior designs. The entire range of vehicle variants is over 10^{100} [84].

Mercedes-Benz used to use a rule-based database where the feasible configurations for a vehicle is stored, as shown in Figure A.8. Each order must undergo this database before being accepted as a contract [141].

For instance, the vehicle model CLA of Mercedes-Benz has several different variants in terms of performance (116, 150, or 190 PS). In this case, the engine control unit is designed based on the performance. To manage variants from the OEM's perspective, a part number is assigned to each type of the motor controller. When a vehicle of a model needs to be produced, the corresponding motor control unit is ordered from the supplier. The resulting configuration constraints on vehicle functionalities thus affect the control unit level. For example, the 116 PS version may not have sufficient power to tow a trailer, leading to the exclusion of the option for a trailer hitch.

The traditional approach of employing a coarse granularity on the level of control units in variant management within the automotive industry has become unpractical, primarily due to the following factors:

- The ECUs can be configured with an increasing level of detail. For instance, the body comfort system can vary in terms of functionalities related to window systems, safety standards, and user interfaces. If variant richness were considered at the control unit level, numerous control units would have to be stored for each variant to ensure the assembly of the vehicle.
- The life cycle of vehicle software has been significantly reduced to approximately one month. Managing unchanged control unit hardware with updated software becomes less meaningful.
- Communication between the vehicle and the OEM's server enables on-demand functionality, allowing the activation and deactivation of various vehicle functions with the given hardware. In this context, coupling vehicle software and hardware is inappropriate for future business operations.

Therefore, OEMs are working to decouple the management of vehicle software and hardware.

Decoupling of automotive software and hardware

The core difference between variant management in the automotive industry and the IT industry lies in the hardware configuration. Car variants are constructed based on utilized sensors, actuators, and setpoint generators, whereas differences between two personal computers typically are usually addressed in qualitative aspects such as CPU computing power and RAM size. Additionally, the interfaces between vehicle components lack standardization due to variant requirements and realization.

Variant dependencies in automotive variant management can be categorized as follows:

- Hardware-hardware dependency: For instance, configuring a rearview camera necessitates the presence of distance sensors at the rear side.

- Software-software dependency: For example, the software of rearview camera requires specific camera driver to operate
- Software-hardware dependency: For example, developing a new image processing algorithm may require control units with sufficient computing power.
- Hardware-software dependency: µCs from certain semiconductor manufacturers may require corresponding firmware to establish the hardware abstraction layer.

A.5.2 Software product line in automotive industry

Software variability is referred as the ability of software systems or artifacts to be efficiently extended, changed, customized or configured [151]. Due to the complexity of software variability, software families instead of single software system are considered in the automotive industry. Against the background, it is desired that commonly used features in software families are developed together, while the special features of corresponding software systems are developed individually. This principle is the fundamental idea of Software Product Line (SPL), who illustrates an approach of software reuse.

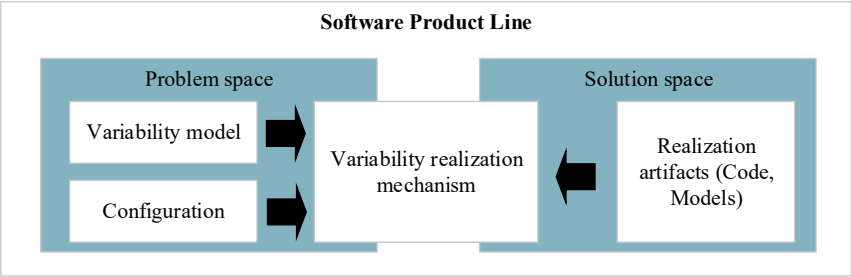


Figure A.9: Software product line in the automotive industry, based on Seidl et al. [138].

In principle, two spaces are used to describe the terminology of SPL, as illustrated in Figure A.9. The problem space presents the information on conceptual level, where the solution to the problem is irrelevant. The solution space, in contrary, encompasses concrete realization of variants of software systems.

A variability model is used to determine valid *configurations* options and to state the feasible combinations of an SPL. Three types of variability models are used: feature models, decision models, and orthogonal variability models [138], in which feature model is currently the most widely used approach for modeling variability [131].

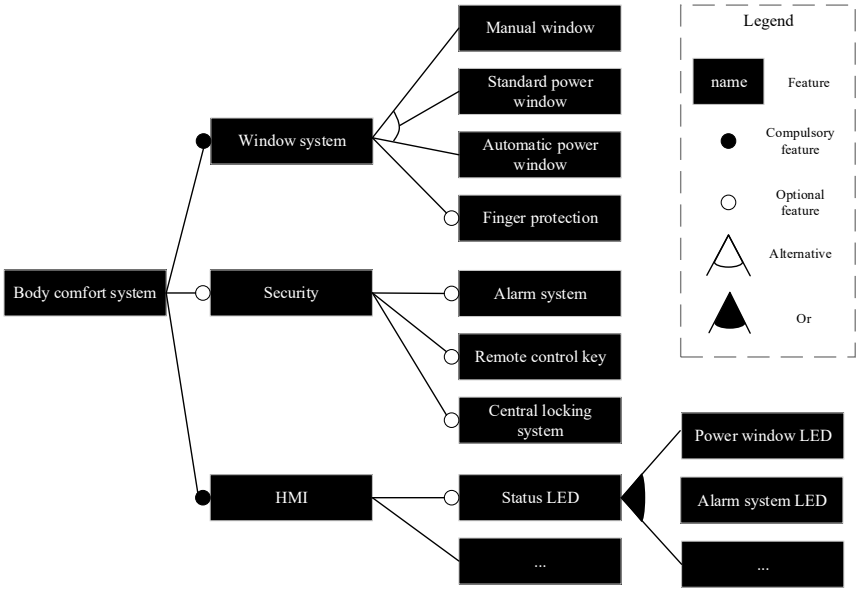


Figure A.10: An example feature model for body system functions, based on Seidl et al. [138].

The solution space is directly connected with realization artifacts, where the SPL is realized by codes or models. Many modeling and programming languages offer variability mechanisms for software variability, including abstract super-classes, implementation interfaces, and conditional compilation [106].

Figure A.10 demonstrates a variability model for body system function using a feature model. Within the body system, three modules are encompassed: the window system and Human Machine Interface (HMI) are compulsory modules, while the security module is optional.

Within the window system, one of the configurations must be chosen (alternative group): Manual window, Standard power window, or Automatic power window. The three types of windows are mutually exclusive, as the windows can be either controlled by motors or by hands.

All functions of Security module are optional: Alarm system, Remote control key, and Central locking system. Within the HMI, Status LED can be chosen to illustrate the status of the window system. Within the HMI field, either the Power window LED or the Alarm system LED, or both can be chosen to build a valid configuration. Additionally, extra rules can be implemented onto the feature model. For instance, if the function remote control key is activated, the function central locking system is then compulsory.

A.5.3 Variability realization

In case that a configuration is chosen for a variant, the solution space shown in Figure A.9 is supposed to derive the artifact for the desired variant. Therefore, realization mechanisms are needed to derive the desired variant from the together developed similarities of the SPL. In principle, variability realization mechanisms can be distinguished into three groups [138]:

- Annotative variability realization mechanism,
- Compositional variability realization mechanism, and
- Transformational variability realization mechanism.

Figure A.11 gives an overview of these variability realization mechanisms. The annotative variability realization mechanism employs so-called 150% model that

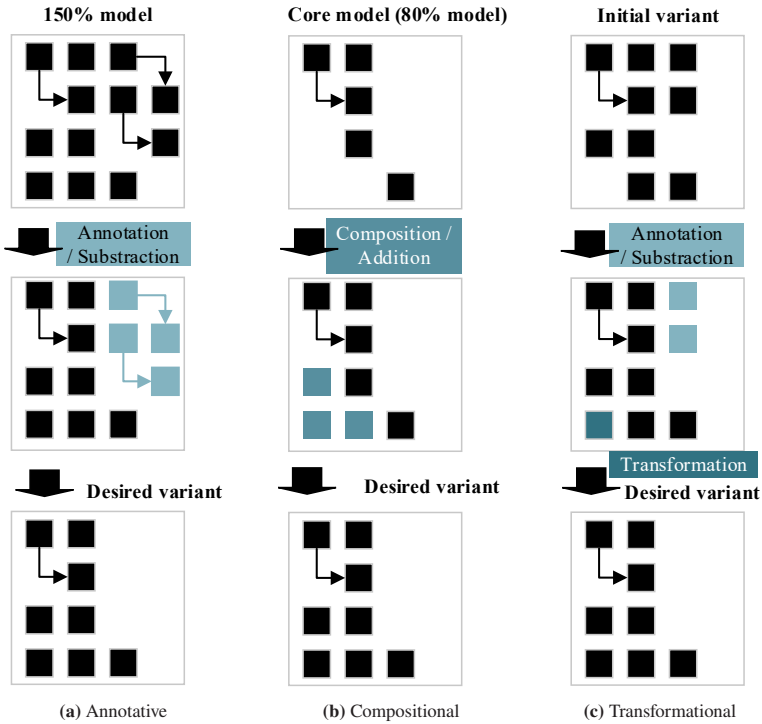


Figure A.11: Variability realization mechanisms, based on Seidl et al. [138].

contains the information of all variants. Based on the 150% model, the desired variant is derived through subtraction/annotation of the software components, as Figure A.11-(a) illustrates. Annotation can be utilized either within realization artifacts (internal annotation) or during the design of the SPL (external annotation) [131].

For example, conditional control flow branches (if... else ...) can be used. In particular, in the programming language C, it is common practice to use the preprocessor with the conditional compilation (#ifdef ... #elseif ... #endif). Compared to other variability realization mechanisms, the annotative approach is shown to be the most widely used solution for software developers in the automotive industry due to the conceptional model and reduced technical requirements.

The following code block illustrates the usage of conditional compilation of the shown body system in Figure A.10 within the programming language C [138].

```
1  bool clsLocked = false;
2
3  #if defined (STANDARD_PW || AUTOMATIC_PW)
4  bool pwEnabled = true;
5  #endif
6
7  void unlock(void)
8  {
9    clsLocked = false;
10   #if defined (STANDARD_PW || AUTOMATIC_PW)
11   pwEnabled = true;
12   #endif
13 }
14
15 void lock(void)
16 {
17   clsLocked = true;
18 }
```

The compositional variability realization mechanism uses the so-called 80% model (or core model) and derives the variant to be used through addition/composition, as depicted in Figure A.11-(b). A concrete implementation of a compositional variability realization mechanism is the so-called Feature-oriented Programming (FOP) [144]. Compared to the annotative variability realization mechanisms, the compositional variability realization mechanism causes higher dissipation across multiple artifacts. Although this mechanism increases the separation of concerns, it may also increase maintenance overhead.

A transformational variability realization mechanism represents variability through a sequence of transformations of an initial variant, as shown in Figure A.11c. The precise actions of the transformations in the variability realization mechanism include adding, removing, and modifying individual elements of a realization artifact. The starting point for the transformation is the initial variant of the SPL, a concrete instantiation of the software system from the software family. The initial

variant is determined manually by developers, and in principle, any valid variant can be used as the initial variant. If certain configuration options are already implemented in the initial variant, a transformation for removal must be provided for subtracting them. Conversely, a transformation for addition must be provided for selecting if certain configuration options are not present in the initial variant.

A.6 Mechanisms for software partitioning in µC-based systems

Regardless of the chosen partitioning approach, partitioned embedded software must be implemented with physical separation in µC-based systems (see Definition 4.5). Depending on the specific µC architecture and the semiconductor manufacturer, different mechanisms can be employed to achieve this separation. Assuming that the basic part and a customer part must be physically separated as defined in Definition 4.5, this section outlines feasible mechanisms currently used in the industry.

A.6.1 Manipulation of linker scripts

A linker script is a file that determines the output sections, orders, types and the address of the first instruction for the compiler [173]. In case that the embedded software is developed only upon low-level driver from semiconductor manufacturer, for instance, *Infineon Low Level Driver* [75], the compilation process can be configured specifically to achieve physical separation.

Assuming the basic part and a customer part are derive and must be compiled by manipulated linker scripts, the source codes of the basic and customer parts can be developed individually, where the function of the customer parts within the main function of the basic part are only declared without realization, see Figure A.12.

The linker script of the basic part arranges functions in the program memory and data in the parameter memory as required (see also the flash memory organization of µCs in Figure 2.14). Within the compilation process of the basic part, the realization of *function_basic_part()* is determined and located in the planned address, while *function_customer_part_A()* is only allocated without realization, serving as a proxy of function realization in the perspective customer part. The individually developed customer part uses another linker script that is dependent of

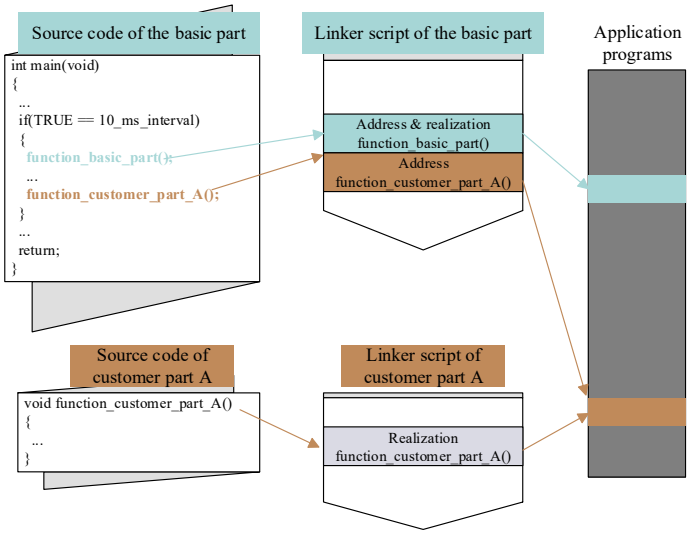


Figure A.12: Example realization of physical separation by manipulation of linker scripts in µC-based systems

the linker script of the basic part. The linker script of customer part A distributes the realization of *function_customer_part_A()* to the desired address.

With a similar principle, allocation of variables and constants of the basic and customer parts can be arranged by the linker scripts, realizing inter-part communication.

We notice that manipulated linker scripts realize software partitioning for software systems with low-level hardware abstraction. However, this kind of manipulation requires profound knowledge of compilation processes of corresponding semiconductor manufacturers. Additionally, the linker script of the basic part cannot know the memory usage of *function_customer_part_A()* during its usage. Therefore, the functionalities of customer parts must be encapsulated in the given function addresses with dedicated calling interval, such as *function_customer_part_A()* in Figure A.12. As the manipulation of output address often violates debug information, the debugging process for the developers can be challenging in case of the usage of manipulated linker scripts.

A.6.2 Software cluster connection of AUTOSAR Classic platform

The concept of Software Cluster Connection (SwCluC) is initially released in 2020 that realizes physical separation within embedded software systems. A *part* of an embedded software system is referred to as *cluster* in SwCluC, see Figure A.13. Within the concept usage, an µC-based system with one BSW stack may have 2 - 20 software clusters, each of which can be individually compiled and flashed [15].

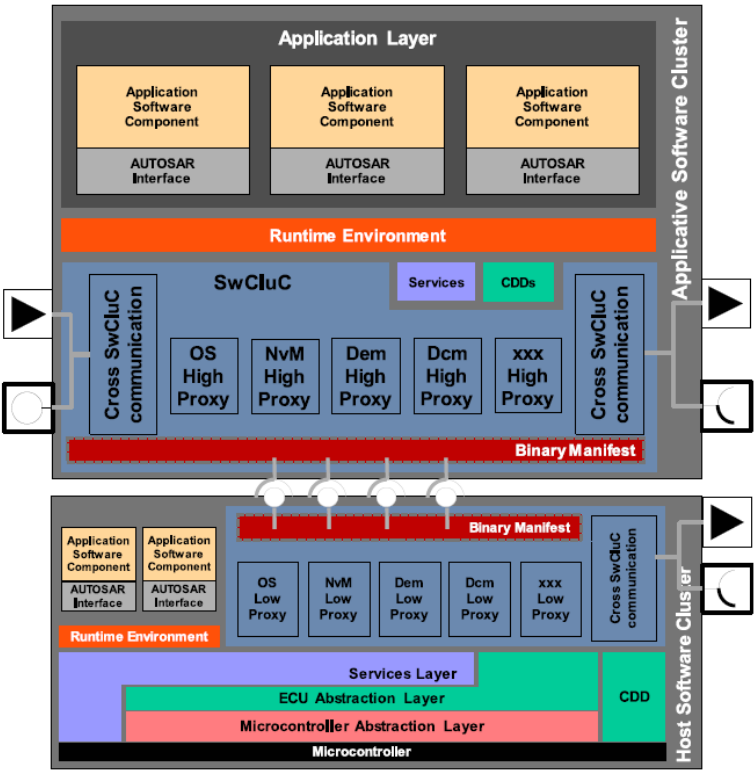


Figure A.13: Software Cluster Connection (SwCluC) in the AUTOSAR Classic Platform (CP), based on AUTOSAR [15]

Within the software cluster concept, the *Host Software Cluster* is the cluster containing the major part of Basic Software (BSW), while the *Applicative Software Cluster* mainly contains SW-Cs. Regarding production-optimized software partitioning, the basic part can be implemented in the form of host software cluster in the software cluster concept. Customer parts can be implemented in the form of applicative software clusters.

The implementation of software clusters within the CP is dependent on the three mechanisms [15]:

- *Binary Manifest* encompasses the start addresses of functions and data in memory of µC-based systems, for instance, the start address of *function_customer_part_A()* in Figure A.12. Additionally, the binary manifest provides handles and interfaces for resources and information to the compiled binary files of software clusters.
- *Cross-cluster Communication* is a realization of *inter-part communication* (Section 4.3.1), in CP methodology, implementation on the level of Virtual Functional Bus (VFB) [19].
- *Proxy Modules* offers applicative clusters with interfaces to BSW modules. The *High Proxy* (see Figure A.13) in the applicative clusters replaces the BSW stack. Correspondingly, the *Low Proxy* in the host cluster supports the High Proxy to realize the functionalities.

Nevertheless, implementing software clusters within the CP methodology presents several challenges [15]:

1. The dynamic behavior of applicative clusters is dependent on the host cluster, with tasks in applicative clusters required to adhere to the specified scheduling constraints.
2. The coupling between clusters should be possibly stable. A change of interface causes the update of the whole software package.
3. Interrupts can hardly be utilized within applicative clusters.

4. Applicative clusters are restricted to using only the BSW modules provided by the high proxies.

A.6.3 Virtualization

Virtualization enjoys immense popularity in the industry as it promises a flexible way to integrate, manage, and re-use heterogeneous software components while obtaining isolation guarantees [41]. The decoupling of virtual and physical computing platforms via Virtual Machine (VM) system supports various functionalities such as resource allocation, load balancing, power management and usage of different operating systems [63].

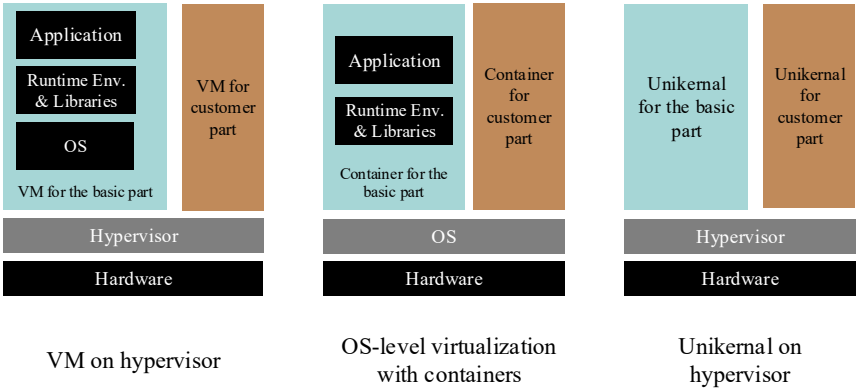


Figure A.14: Examples of virtualization approaches, based on Cinque et al. [41].

Different approaches can be used to implement virtualization on embedded systems, see Figure A.14. Notably, usage of VM on embedded systems requires support from semiconductor manufacturers. The embedded Linux may not work with 8- or 16-bit µC systems. For embedded software systems leveraging virtualization, challenges regarding resource consumption requirements need to be addressed specifically, as the middleware for realization of VMs, such as OS and hypervisor, leads to extra memory and computing power consumption.

For instance, the substantial computational resources required by hypervisors often necessitate the use of higher-performance and more expensive hardware platforms (e.g., Infineon TC4xx series [48]), which may not be cost-effective for many standard automotive ECUs targeting specific, less complex functions. Additionally, without partitioning at the SW-C level, a hypervisor-based approach would necessitate an additional middleware and a dedicated scheduling mechanism to manage communication and execution across virtual domains, further increasing system complexity.

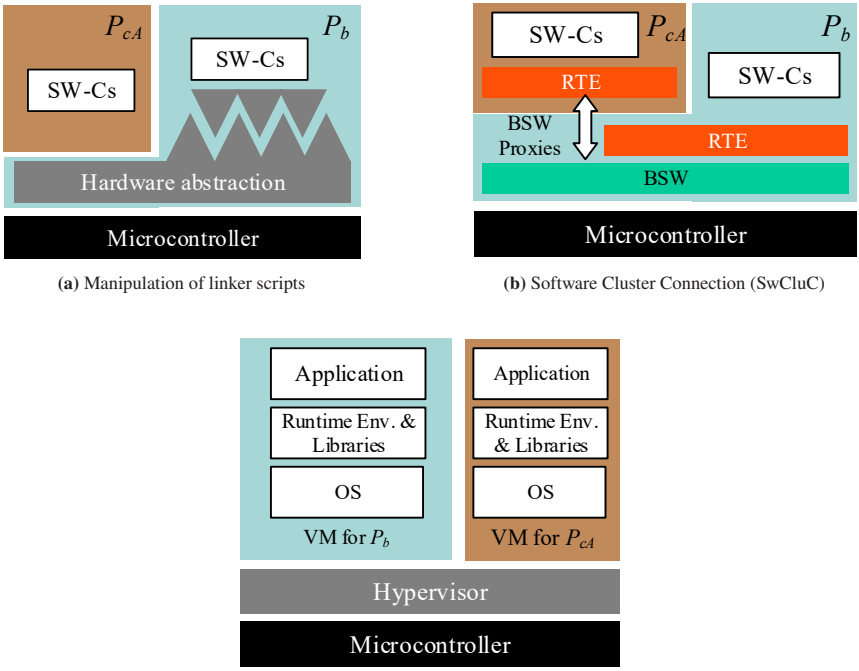
For the use cases of virtualization onto µC-based automotive embedded systems, the following critical limitations are reported regarding the usage of the hypervisor [63]:

- *Scheduling:* The scheduling by the hypervisor is a black box and can hardly guarantee the prioritized tasks in different VMs or real-time operating systems to be carried out;
- *Energy management:* The management of energy in embedded systems cannot be organized locally, which is paradoxical to cars' significant power consumption topic.

A.6.4 Analysis of partitioning mechanisms

A software system partitioned according to POSP (see Definition 4.6) is designed to operate on an µC-based system. Within this context, the realization of a partitioned software system on prototypes can be achieved using the following mechanisms, as detailed in Section A.6:

- Manipulation of linker scripts (discussed in Section A.6.1 and illustrated in Figure A.15-(a))
- Software Cluster Connection (SwCluC) of the CP (discussed in Section A.6.2 and showed in Figure A.15-(b))
- Virtualization (explained in Section A.6.3 and depicted in Figure A.15-(c))



(a) Manipulation of linker scripts

(b) Software Cluster Connection (SwCluC)

(c) Partitioning using virtual machine

Figure A.15: Realization mechanisms to partition software parts physically in µC-based systems

The mechanisms introduced above are analyzed and evaluated based on the following four criteria:

- *Development cost:* considering the cost associated with developing the software solution;
- *Acquisition cost:* referring to the expense of demanded hardware;
- *Real-time ability:* referring to the cost of ensuring the software meets the real-time requirements (see also Section 2.3.1);
- *Reusability:* assessing the effort required to adapt the developed software for use on other hardware platforms.

Each criterion is evaluated using a grading system: $++$, $+$, $-$, and $--$. These symbols represent the impact of the approach in the respective category, with $++$ indicating a very positive effect, $+$ a positive effect, $-$ a negative effect, and $--$ a very negative effect.

Regarding development cost, SwCluC is built on AUTOSAR, which corresponds to significant realization costs [11]. Consequently, it leads to the highest development cost and is rated as $--$. Both manipulated linker scripts and virtualization involve technical challenges, leading to development costs being rated as $-$ for these approaches.

Table A.1: Analysis of partitioning mechanisms

Criterion	**Manipulated linker script**	**SwCluC**	**Virtualization**
Development cost	$-$	$--$	$-$
Acquisition cost	$++$	$+$	$-$
Real-time ability	$++$	$++$	$-$
Reusability	$--$	$+$	$++$
Summary	1	2	-1

The acquisition cost depends on the required hardware capabilities. Manipulated linker scripts can be implemented on a wide range of hardware, making it the most flexible option and earning a grade of $++$. In contrast, SwCluC, built on AUTOSAR, requires the µC to be AUTOSAR-compliant, resulting in a moderate acquisition cost, graded as $+$. Virtualization necessitates additional computing power and upgraded hardware, which raises acquisition costs, leading to a grade of $-$.

In terms of real-time ability, both manipulated linker scripts and SwCluC are developed upon the embedded middleware, ensuring real-time performance without significant technical barriers. Therefore, they are both graded as $++$. However,

virtualization induces challenges in guaranteeing real-time performance within VMs due to the use of hypervisors and OS, which makes it less suitable for real-time applications, resulting in a grade of $-$.

For reusability, manipulated linker scripts are difficult to adapt to hardware from different semiconductor manufacturers, resulting in a grade of $--$. AUTOSAR-compliant software can be reused on other hardware platforms with reconfiguration of Basic Software (BSW), leading to a moderate cost of SwCluC and a grade of $+$. Virtualization offers high flexibility, allowing software to be transferred across platforms as long as the same platform is used, earning a grade of $++$.

By assigning one point to each $+$ grade, the analysis of the discussed approaches can be quantified and compared on a comprehensive basis. This comparison is summarized in Table A.1. The results indicate that SwCluC stands out as a suitable solution for POSP for automotive series software.

Based on these findings, the prototype in this thesis is conducted under the assumption that SwCluC is employed as the partitioning mechanism for automotive software systems.

A.7 Generation of software in the prototype

As discussed in Section 6.1, software of the system under test is built based on the CP, whereas its SW-Cs at the application layer must be designed and the middleware must be configured, see also Section A.2.

More precisely, the boundary conditions of the system under test are given in the form of communication setup files (e.g., communication matrix) and diagnostic setup files. These files regulate the input and output signals of the software system, from which the SW-C at the application layer can be designed.

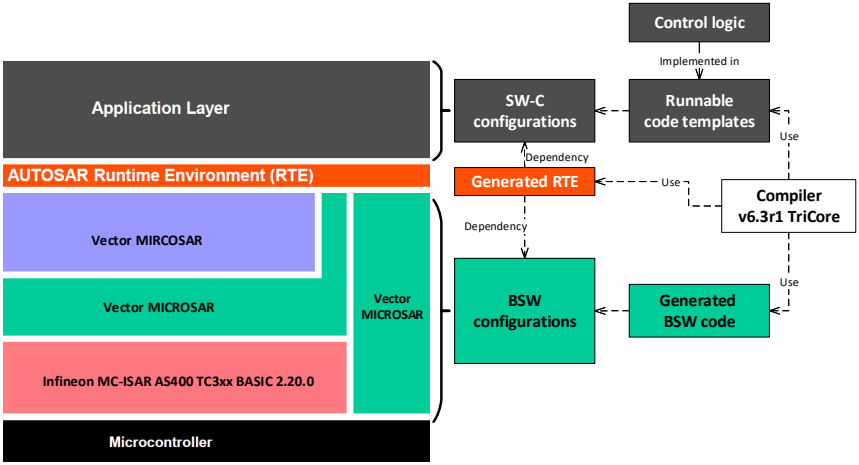


Figure A.16: Generation of system under test software, cf. the CP layered architecture shown in Figure 3.9.

The implementation within the CP relies on several processes and vendors, where every single process must be correct in order to produce executable files, as shown in Figure A.16. The configuration process of BSW is tailored to the hardware of the evaluation board. MICROSAR [160] is used to configure the Services, ECU Abstraction, and Complex Drivers, cf. Figure 3.9. The Microcontroller Abstraction Layer was configured using *Infineon MC-ISAR AS440 TC3xx BASIC 2.20.0* [78].

Based on the BSW and SW-Cs, the RTE is then generated. This is followed by the generation of source code for the BSW, RTE, and Runnable templates within the SW-Cs.

Once the control logic within Runnables is implemented in the generated templates, the system under test can be compiled into an executable binary using the *v6.3r1 TriCore C Compiler* [152], as defined in Definition 3.7.

Acronyms

µC Microcontroller. 7, 8, 10, 26, 28, 34, 35, 37, 47–49, 51, 53, 54, 60, 64, 65, 68, 71, 75, 83–86, 88–90, 93–95, 98–100, 105, 109, 110, 116, 119, 122, 124, 128, 131, 132, 134, 149, 150, 153–156, 165, 176, 185, 186, 191, 192, 194, 200, 202–204, 218, 224–231, 240, 241, 243

ABS Antilock-Braking System. 25, 86

ADAS Advanced Driver-Assistance Systems. 1, 30, 56

ADL Architecture Description Language. 65, 68

AP AUTOSAR Adaptive Platform. 60–64, 75, 85, 193, 240

API Application Interface. 62, 64

ARA AUTOSAR Runtime for Adaptive Applications. 61, 62

AUTOSAR AUTomotive Open System Architecture. 9, 59, 60

BDD Block Definition Diagram. 65–67, 240

BiW Body-in-White. 12–14

BLDC Brushless DC. 145, 241

BSW Basic Software. 60, 62, 71, 74, 78, 84, 156, 161, 168, 169, 172, 186, 193, 200, 202, 203, 226–228, 232–234

CP AUTOSAR Classic Platform. i, ii, 7, 8, 10, 53, 55, 60–64, 68–72, 74–76, 78, 79, 84, 85, 94, 99, 118, 120, 153, 155–157, 160, 165, 185, 186, 191, 193, 200, 201, 203, 204, 207, 208, 226, 227, 229, 233, 240–243

DCM Diagnostic Communication Manager. 78

DEM Diagnostic Event Manager. 78

DMD Door Module Driver. 66, 67, 240

DTC Diagnostic Trouble Code. 31–33

E/E Electrical and/or Electronic. 2, 8, 9, 15, 24–28, 31, 32, 54, 63, 127, 172, 189, 196–198

ECU Electronic Control Unit. 1–3, 6, 21, 27, 28, 31, 33, 35, 38, 39, 47, 50, 53, 59, 61–64, 66, 68, 69, 71, 75, 77, 78, 83–86, 88, 93, 94, 98, 104, 111, 113, 115, 116, 118, 119, 121, 122, 124–127, 129, 130, 132, 136, 150, 154, 156, 167, 173, 176, 178, 186, 192–194, 201, 202, 204, 210, 215, 217, 240–242

EEPROM Electrically Erasable Programmable ROM. 35

HPC High Performance Computer. 28, 32, 51, 60, 63, 64

IBD Internal Block Diagram. 65–67, 240

MBSE Model-Based Systems Engineering. 64, 65, 71

OBD On-Board Diagnostics. 32, 33

OEM Original Equipment Manufacturer. 1, 3, 11, 18, 21, 22, 31, 41, 59, 60, 64, 68, 69, 77, 83, 95, 189, 190, 203, 205

OS Operating System. 228

OTA Over the Air. 31, 63

PDU Protocol Data Unit. 58, 59

PORaSPA Production-Oriented Rundundancy-bAsed Software Partitioning Approach. 138, 140, 150

POSIX Portable Operating System Interface. 61–63

POSP Production-Oriented Software Partitioning. 10, 95–98, 100, 101, 103, 106, 109, 111–115, 118–120, 122–127, 129, 133, 136, 148, 151, 154, 156, 165, 170, 172, 173, 176, 178, 180, 183, 185, 186, 190–194, 229, 232, 241

RAM Random-Access Memory. 34, 35, 212

RFLP Requirement, Functional, Logical, and Physical. 22, 24, 26, 68, 72

ROM Read-Only Memory. 34, 35, 47–49, 212, 240

RTE Run Time Environment. 60, 84, 99, 111, 120, 169, 172, 193, 200, 202, 234

S/R Sender or Receiver. 101, 102

SOP Start of Production. 2, 72, 76, 171

SPICE Software Process Improvement and Capability dEtermination. 18–23, 239

SW-C Software Component. ix, 10, 16, 21, 26, 27, 50–54, 56, 57, 60, 61, 66, 67, 69–72, 74, 75, 78–80, 84, 85, 87, 88, 90, 94, 97–103, 105–109, 112, 113, 119, 121, 122, 124–127, 129, 130, 132, 134, 138–150, 161–182, 186, 187, 191–193, 200–203, 206, 208, 209, 227, 233, 234, 241, 242

SwCluC Software Cluster Connection. 153, 156, 163, 165, 185, 191, 193, 194, 226, 229–232, 243

UDS Unified Diagnostics Services. 33, 37, 40, 159

VFB Virtual Functional Bus. 201–203, 227, 242

VM Virtual Machine. 228, 232

VSA Vehicle Systems Architecture. 22–24, 26, 27, 50, 84, 85, 87, 89, 239

List of Figures

List of Tables

Publications

[Z1] Aiman El Asad, Michael Hahn, Yi Zhai, and Hans-Christian Reuss. Advancing automotive production: An llm-based impact analysis for software updates. *Procedia CIRP*, 134:127–132, 2025. 58th CIRP Conference on Manufacturing Systems 2025. URL: https://www.sciencedirect.com/science/article/pii/S2212827125004639, doi:10.1016/j.procir.2025.02.134.

[Z2] Dennis Griebler, Yi Zhai, Mario Caggiano, and Thomas Fuchss. Enhancing flexibility of automotive embedded software: A modular reference architecture. In *2025 Stuttgart International Symposium*. SAE International, July 2025. doi:10.4271/2025-01-0282.

[Z3] Yi Zhai, Maximilian Beck, Aiman El Asad, Katja Köhler, Michael Hahn, and Eric Sax. Production-optimized automotive software architecture: Theory and applications. *IEEE Transactions on Industrial Informatics*, 2026. doi:10.1109/TII.2026.3651904.

[Z4] Yi Zhai, Michael Hahn, Mario Caggiano, and Eric Sax. Automotive software partitioning: A production-centric perspective. In *Proceedings of the 4th European Symposium on Software Engineering*, ESSE '23, page 23–30, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3651640.3651647.

[Z5] Yi Zhai, Andreas Vetter, and Eric Sax. Analysis of current challenges of automotive software in the view of manufacturing. In *23rd Stuttgart International Symposium*. SAE International, jun 2023. doi:10.4271/2023-01-1221.

[Z6] Yi Zhai, Moritz Zink, Houssem Guissouma, Michael Hahn, Mario Caggiano, and Eric Sax. Analysis of update capabilities of modern automotive electric/electronic architectures. In *2024 IEEE 19th Conference on Industrial Electronics and Applications (ICIEA)*, pages 1–8, 2024. doi:10.1109/ICIEA61579.2024.10665021.

Bibliography

[1] Mesfin Abebe and Cheol-Jung Yoo. Trends, opportunities and challenges of software refactoring: A systematic literature review. 2014. URL: https://api.semanticscholar.org/CorpusID:30990088.

[2] ACEA. Automobile industry pocket guide 2022-2023, September 2022. URL: https://www.acea.auto/publication/automobile-industry-pocket-guide-2022-2023/.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[4] Louis Altazan. Diagnosing power window problems, 2023. URL: https://www.agcoauto.com/content/news/p2_articleid/153.

[5] A. Aly, E. Zeidan, A. Hamed, and F. Salem. An antilock-braking systems (ABS) control: A technical review. *Intelligent Control and Automation*, 2(3):186–195, 2011. URL: https://www.scirp.org/journal/paperinformation?paperid=6667, doi:10.4236/ica.2011.23023.

[6] Amazon Web Services. Was ist der Unterschied zwischen SOA und Microservices, December 2023. URL: https://aws.amazon.com/de/compare/the-difference-between-soa-microservices/.

[7] Amphenol Thermometrics, Inc. *Sensor Temperature Resistance Curves*. Amphenol Thermometrics, Inc., 967 St. Marys, PA 15857-3333, USA, b edition, September 2014. URL: https://www.amphenol-

sensors.com/hubfs/Documents/AAS-913-318B-Temp-Resistance-Curves-091614-web.pdf.

[8] Vard Antinyan. Revealing the Complexity of Automotive Software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1525, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3368089.3417038, doi:10.1145/3368089.3417038.

[9] Andreas Antoniou and Wu-Sheng Lu. *The Optimization Problem*, pages 1–26. Springer US, New York, NY, 2021. doi:10.1007/978-1-0716-0843-2_1.

[10] Anna Arestova, Maximilian Martin, Kai-Steffen Jens Hielscher, and Reinhard German. A service-oriented real-time communication scheme for autosar adaptive using opc ua and time-sensitive networking. *Sensors*, 21(7), 2021. URL: https://www.mdpi.com/1424-8220/21/7/2337, doi:10.3390/s21072337.

[11] David Arthur, Christopher Becker, Alex Epstein, Bill Uhl, and Scott Ranville. *Foundations of Automotive Software*, 2022. URL: https://rosap.ntl.bts.gov/view/dot/62489.

[12] Atmel Corporation. *AVR236: CRC Check of Program Memory*. Atmel Corporation, April 2002. URL: https://ww1.microchip.com/downloads/en/Appnotes/doc1143.pdf.

[13] AUTOSAR. *Specification of Diagnostic Communication Manager*. AUTOSAR, 12 2017. URL: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_DiagnosticCommunicationManager.pdf.

[14] AUTOSAR. *Specification of Diagnostic Event Manager*. AUTOSAR, 12 2017. URL: https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_SWS_DiagnosticEventManager.pdf.

[15] AUTOSAR. *Explanation of CP Software Cluster Design And Integration Guideline*. AUTOSAR, 11 2020. URL: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_CPSwClusterDesignAndIntegrationGuideline.pdf.

[16] AUTOSAR. *Explanation of Firmware Over-The-Air*, November 2020. URL: https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_EXP_FirmwareOverTheAir.pdf.

[17] AUTOSAR. *Explanation of Adaptive Platform Software Architecture*, November 2021. URL: https://www.autosar.org/fileadmin/standards/R21-11/AP/AUTOSAR_EXP_SWArchitecture.pdf.

[18] AUTOSAR. *Software Component Template*. AUTOSAR, November 2021. URL: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_TPS_SoftwareComponentTemplate.pdf.

[19] AUTOSAR. *Specification of Software ClusterConnection module*. AUTOSAR, 11 2021. URL: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_SoftwareClusterConnection.pdf.

[20] AUTOSAR. *Application Interfaces UserGuide*, November 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_AIUserGuide.pdf.

[21] AUTOSAR. *Explanation of Adaptive PlatformDesign*. AUTOSAR, November 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_EXP_PlatformDesign.pdf.

[22] AUTOSAR. *Layered Software Architecture*. AUTOSAR, November 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[23] AUTOSAR. *Methodology for Classic Platform*. AUTOSAR, November 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_TR_Methodology.pdf.

[24] AUTOSAR. *Classic Platform*. AUTOSAR, November 2023. URL: https://www.autosar.org/standards/classic-platform.

[25] AUTOSAR. *Foundation Release Overview*. AUTOSAR, November 2023. URL: https://www.autosar.org/fileadmin/standards/R23-11/FO/AUTOSAR_FO_TR_ReleaseOverview.pdf.

[26] Fabrice Auzanneau. Wire troubleshooting and diagnosis: Review and perspectives. *Progress In Electromagnetics Research B*, 49:253–279, 01 2013. doi:10.2528/PIERB13020115.

[27] Mohd Azri, Azri A. Aziz, M. Saifizi Saidon, M Izuan Fahmi, Siti Othman, Wan Mustafa, Mohd Rizal Manan, and Muhammad Aihsan. A review on bldc motor application in electric vehicle (ev) using battery, supercapacitor and hybrid energy storage system: Efficiency and future prospects. *Journal of Advanced Research in Applied Sciences and Engineering Technology*, 30:41–59, 04 2023. doi:10.37934/araset.30.2.4159.

[28] Victor Bandur, Gehan Selim, Vera Pantelic, and Mark Lawford. Making the Case for Centralized Automotive E/E Architectures. *IEEE Transactions on Vehicular Technology*, 70(2):1230, 2021. doi:10.1109/TVT.2021.3054934.

[29] Abdulrahman Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28, 06 2020. doi:10.1007/s11219-019-09477-y.

[30] Jan Bauer and Markus Kreuzer. Understanding the requirements for automotive displays in ambient light conditions. *Information Display*, 32(3):14–22, 2016. URL: https://sid.onlinelibrary.wiley.com/doi/abs/10.1002/j.2637-496X.2016.tb00902.x, doi:10.1002/j.2637-496X.2016.tb00902.x.

[31] Kevin Baughey. Functional and logical structures: A systems engineering approach. In *SAE 2011 World Congress & Exhibition*. SAE International, April 2011. doi:10.4271/2011-01-0517.

[32] Sandra Bickelhaupt, Michael Hahn, Nikolai Nuding, Andrey Morozov, and Michael Weyrich. Challenges and opportunities of future vehicle diagnostics in software-defined vehicles. In *WCX SAE World Congress Experience*. SAE International, April 2023. doi:10.4271/2023-01-0847.

[33] BlackBerry Limited. Blackberry qnx automotive software, December 2023. URL: https://blackberry.qnx.com/en/industries/connected-autonomous-vehicles.

[34] BlackBerry Limited. Microkernel architecture, December 2023. URL: https://blackberry.qnx.com/en/ultimate-guides/what-is-real-time-operating-system/microkernel-architecture.

[35] Daniel Bogdan, Razvan Bogdan, and Mircea Popa. Delta flashing of an ecu in the automotive industry. In *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 503–508, 2016. doi:10.1109/SACI.2016.7507429.

[36] Kai Borgeest. *Bordelektrik*, pages 3–23. Springer Fachmedien Wiesbaden, Wiesbaden, 2014. doi:10.1007/978-3-8348-2145-4_2.

[37] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 33–42, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1134285.1134292.

[38] Manfred Broy, Ingolf Kruger, Alexander Pretschner, and Christian Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95:356, March 2007. doi:10.1109/JPROC.2006.888386.

[39] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5–es, feb 2007. doi:10.1145/1189748.1189753.

[40] Stefan Brunner, Jurgen Roder, Markus Kucera, and Thomas Waas. Automotive e/e-architecture enhancements by usage of ethernet tsn. In *2017 13th*

Workshop on Intelligent Solutions in Embedded Systems (WISES), pages 9–13, 2017. doi:10.1109/WISES.2017.7986925.

[41] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21004787, doi:10.1016/j.future.2021.12.002.

[42] Constanze Clarke. *Automotive Production Systems and Standardisation*. Physica Heidelberg, March 2006. URL: https://link.springer.com/book/10.1007/b138988, doi:10.1007/b138988.

[43] Yanja Dajsuren. *On the design of an architecture framework and quality evaluation for automotive software systems*. phdthesis, Eindhoven University of Technology, May 2015. URL: https://pure.tue.nl/ws/portalfiles/portal/15934981/20160307_Dajsuren.pdf.

[44] Yanja Dajsuren and Mark van den Brand. *Automotive Software Engineering: Past, Present, and Future*, pages 3–8. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-12157-0_1.

[45] Marco Di Natale and Alberto Luigi Sangiovanni-Vincentelli. Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603, 2010. doi:10.1109/JPROC.2009.2039550.

[46] dSPACE GmbH. *TargetLink: Code-Generierung für höchste Anforderungen*. dSPACE GmbH, 2022. URL: https://www.dspace.com/de/gmb/home/products/sw/pcgs/targetlink.cfm.

[47] EAST-ADL Association. *About EAST-ADL*. EAST-ADL Association, 2023. URL: https://www.east-adl.info/Specification.html.

[48] Elektrobit. Elektrobit announces first automotive-grade, embedded OS and hypervisor for infineon AURIX TC4x microcontrollers, Oct 2022. [Online].

URL: https://www.elektrobit.com/newsroom/first-automotive-grade-embedded-os-and-hypervisor/.

[49] Elektrobit. Classic autosar with eb tresos, January 2024. URL: https://www.elektrobit.com/products/ecu/eb-tresos/.

[50] Embitel. How Different are On-board and Off-board Vehicle Diagnostics? A Detailed Analysis of Functions, Scope and Services, June 2018. URL: https://www.embitel.com/blog/embedded-blog/off-board-vs-on-board-vehicle-diagnostics-everything-you-ever-wished-to-know.

[51] M.C. Feathers. *Working Effectively with Legacy Code*. Martin, Robert C. Prentice Hall Professional Technical Reference, 2004. URL: https://books.google.de/books?id=vlo_nWophSYC.

[52] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis and Design Language (AADL):An Introduction*, February 2006. URL: https://www.sei.cmu.edu/documents/2103/2006_004_001_14678.pdf.

[53] FIA Region I. The automotive digital transformation: Study on software updates and aftermarket. https://www.fiaregion1.com/wp-content/uploads/2023/09/FIA-The-automotive-digital-transformation-STUDY-FINAL.pdf, September 2023. Accessed: 2025-11-17.

[54] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani. *Software Architecture: The Hard Parts*. O'Reilly Media, October 2021. URL: https://www.oreilly.com/library/view/software-architecture-the/9781492086888/.

[55] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML (Second Edition)*. Morgan Kaufmann, Boston, second edition, 2012. URL: https://www.sciencedirect.com/science/article/pii/B9780123852069110014, doi:10.1016/B978-0-12-385206-9.11001-4.

[56] Umberto Fugiglando, Emanuele Massaro, Paolo Santi, Sebastiano Milardo, Kacem Abida, Rainer Stahlmann, Florian Netter, and Carlo Ratti. Driving behavior analysis through can bus data in an uncontrolled environment. *IEEE Transactions on Intelligent Transportation Systems*, 20(2):737–748, 2019. doi:10.1109/TITS.2018.2836308.

[57] Paolo Gai, Moisés Urbina, Errico Guidieri, Giuseppe Serano, and Nicola Serreli. Autosar university package classic platform. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, 2022. doi:10.1109/ETFA52439.2022.9921596.

[58] Thomas M. Galla, Dietmar Schreiner, Wolfgang Forster, Christof Kutschera, Karl M. Goschka, and Martin Horauer. Refactoring an automotive embedded software stack using the component-based paradigm. In *2007 International Symposium on Industrial Embedded Systems*, pages 200–208, 2007. doi:10.1109/SIES.2007.4297336.

[59] Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, George Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1):39–58, 2008. Special Issue on Foundations and Applications of Model Driven Architecture (MDA). URL: https://www.sciencedirect.com/science/article/pii/S0167642308000518, doi:10.1016/j.scico.2008.05.005.

[60] Houssem Guissouma, Carl Philipp Hohl, Fabian Lesniak, Marc Schindewolf, Jürgen Becker, and Eric Sax. Lifecycle management of automotive safety-critical over the air updates: A systems approach. *IEEE Access*, 10:57696–57717, 2022. doi:10.1109/ACCESS.2022.3176879.

[61] Houssem Guissouma, Heiko Klare, Eric Sax, and Erik Burger. An empirical study on the current and future challenges of automotive software release and configuration management. In *2018 44th Euromicro Conference on*

Software Engineering and Advanced Applications (SEAA), pages 298–305, 2018. doi:10.1109/SEAA.2018.00056.

[62] Waldemar Haas and P. Langjahr. Cross-domain vehicle control units in modern e/e architectures. In Michael Bargende, Hans-Christian Reuss, and Jochen Wiedemann, editors, *16. Internationales Stuttgarter Symposium*, pages 1619–1627, Wiesbaden, 2016. Springer Fachmedien Wiesbaden.

[63] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, page 11–16, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1435458.1435461.

[64] B. Henderson and J. Haynes. *OBD-II & Electronic Engine Management Systems*. Haynes Techbook. Haynes Manuals N. America, Incorporated, 2006. URL: https://books.google.de/books?id=wsuPPQAACAAJ.

[65] Jacqueline Henle, Mona Gierl, Houssem Guissouma, Felix Müller, Goutham Bharadwaj Ramesh, and Eric Sax. Concept for an approval-focused over-the-air update development process. In *23rd Stuttgart International Symposium*. SAE International, jun 2023. doi:10.4271/2023-01-1224.

[66] Jacqueline Henle, Martin Stoffel, Marc Schindewolf, Ann-Therese Nägele, and Eric Sax. Architecture platforms for future vehicles: a comparison of ros2 and adaptive autosar. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 3095–3102, 2022. doi:10.1109/ITSC55140.2022.9921894.

[67] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 1–15, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[68] Bernd Hindel and Bernhard Sechser. Qualitätssicherung für die software im auto mit automotive spice 3.0. techreport, Heise Medien, December 2015. URL: https://www.heise.de/hintergrund/Qualitaetssicherung-

fuer-die-Software-im-Auto-mit-Automotive-SPICE-R-3-0-3029717.html?seite=2.

[69] Frank Houdek and Barbara Paech. Das Türsteuergerät – eine Beispielspezifikation. resreport, DaimlerChrysler, January 2002. URL: https://publica.fraunhofer.de/entities/publication/4ff6f36c-ea7f-4aff-99ec-1e862181ba49.

[70] Shamsul Huda, Sultan Alyahya, Mohsin Ali, Shafiq Ahmad, Jemal Abawajy, Hmood Al-Dossari, and John Yearwood. A framework for software defect prediction and metric selection. *IEEE Access*, 6:2844–2858, 12 2017. doi:10.1109/ACCESS.2017.2785445.

[71] IBM Cloud Team. Soa vs. microservices: What's the difference?, May 2021. URL: https://www.ibm.com/blog/soa-vs-microservices/.

[72] ifm electronic. Processes in theautomotive industry, January 2016. URL: https://www.ifm.com/binaries/content/assets/pdf-files/en/catalogues/ifm-automation-technology-automotive-industry-catalogue-2019-2020.pdf.

[73] INCOSE. *Systems Engineering Vision 2020*. INCOSE, September 2007. INCOSE-TP-2004-004-02. URL: https://sebokwiki.org/wiki/INCOSE_Systems_Engineering_Vision_2020.

[74] Infineon Technologies AG. Infineon Technologies AURIX™ TC3xx Microcontrollers. techreport, Infineon Technologies AG, October 2016. URL: https://www.mouser.de/new/infineon/infineon-aurix-tc3xx-mcus/?srsltid=AfmBOopr7H1yGSOu8t0R8vPkyb4qTCqUxm3so_cPIGNNgn0pxEWHSEDV.

[75] Infineon Technologies AG. *iLLD - Infineon Low Level Driver*, March 2019. URL: https://www.infineon.com/dgdl/Infineon-AURIX_Infineon_Low_Level_Driver-Training-v01_00-EN.pdf?fileId=5546d46269bda8df0169ca77502b254c.

[76] Infineon Technologies AG. *KIT_A2G_TC397_3V3_TFT – AURIX™ TC397 Evaluation Board*, 2019. User Manual Version 02.00, retrieved June 21, 2025. URL: https://www.infineon.com/cms/en/product/evaluation-boards/kit_a2g_tc397_3v3_tft/.

[77] Infineon Technologies AG. *Flash Programming for KIT AURIX TC397 TFT*. Infineon Technologies AG, 81726 Munich, Germany, June 2020. URL: https://www.infineon.com/dgdl/Infineon-AURIX_Flash_Programming_1_KIT_TC397_TFT-Training-v01_00-EN.pdf?fileId=5546d46272e49d2a0172e6e11d5c01f8.

[78] Infineon Technologies AG. *AURIX™ Embedded Software – AUTOSAR MCAL Drivers – MC-ISAR AURIX™*. Infineon Technologies AG, 2025. Accessed: 2025-07-02. URL: https://www.infineon.com/design-resources/platforms/aurix-software-tools/aurix-embedded-sw/autosar.

[79] ISO Central Secretary. Road vehicles — Open diagnostic data exchange (ODX) — Part 1: Data model specification. Standard ISO 22901-1:2008, ISO, Geneva, CH, 2008. URL: https://www.iso.org/standard/41207.html.

[80] ISO Central Secretary. ISO/IEC 25010:2011 - Systems and software Quality Requirements and Evaluation (SQuaRE). standard, ISO, Geneva, CH, 2017. URL: https://www.iso.org/standard/78175.html.

[81] ISO Central Secretary. ISO/IEC 33020:2019 - Information technology — Process assessment — Process measurement framework for assessment of process capability. standard, Geneva, CH, 2019. URL: https://www.iso.org/standard/78526.html.

[82] ISO Central Secretary. Road vehicles — Unified diagnostic services (UDS) — Part 1: Application layer. Standard, ISO, Geneva, CH, February 2020. URL: https://www.iso.org/standard/72439.html.

[83] Shugang Jiang. Vehicle e/e architecture and its adaptation to new technical trends. *SAE Technical Paper 2019-01-0862*, April 2019. doi:10.4271/2019-01-0862.

[84] Florian Jost and Carsten Sinz. Handling automotive hardware/software co-configurations with integer difference logic. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '24, page 103–111, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3634713.3634728.

[85] Ulrich Jürgens. *Automatisierung und Arbeit in der Automobilindustrie: Von Henry Ford zur Industrie 4.0*. Nomos Verlagsgesellschaft mbH & Co. KG, Baden-Baden, 2023. doi:10.5771/9783748929055.

[86] Alexandru Kampmann, Bassam Alrifaee, Markus Kohout, Andreas Wüstenberg, Timo Woopen, Marcus Nolte, Lutz Eckstein, and Stefan Kowalewski. A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2101–2108, 2019. doi:10.1109/ITSC.2019.8916841.

[87] Sebastian Kiebusch, Bogdan Franczyk, and Andreas Speck. Process-family-points. In Qing Wang, Dietmar Pfahl, David M. Raffo, and Paul Wernick, editors, *Software Process Change*, pages 314–321, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[88] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt.verlag, 2009. URL: https://books.google.de/books?id=dHF4DwAAQBAJ.

[89] Osamu Kitazawa, Takaji Kikuchi, Masaru Nakashima, Yoshiki Tomita, Hajime Kosugi, and Takahisa Kaneko. Development of power control unit for compact-class vehicle. *SAE International Journal of Alternative Powertrains*, 5(2):278–285, 2016. URL: http://www.jstor.org/stable/26169132.

[90] Jonas Krog, Tarık Şahin, and Thomas Vietor. Towards a systems engineering methodology for architecture development of vehicle concepts.

In *Proceedings of NordDesign 2022*, pages 12–12, January 2022. doi:10.35199/NORDDESIGN2022.35.

[91] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12:45–50, 11 1995. doi:10.1109/52.469759.

[92] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012. doi:10.1109/MS.2012.167.

[93] Stefan Kugele, Philipp Obergfell, Manfred Broy, Oliver Creighton, Matthias Traub, and Wolfgang Hopfensitz. On Service-Orientation for Automotive Software. In *2017 IEEE International Conference on Software Architecture (ICSA)*, page 193, 2017. doi:10.1109/ICSA.2017.20.

[94] Stefan Kugele, Philipp Obergfell, and Eric Sax. Model-based resource analysis and synthesis of service-oriented automotive software architectures. *Software and Systems Modeling*, 20(6):1945–1975, Dec 2021. doi:10.1007/s10270-021-00896-9.

[95] H. Kühnle. *Lecture Notes in Manufacturing Systems Design and Manufacturing Process Organisation: Selected Chapters from Factory Operations, Factory Planning, Manufacturing Enterprise Organisation & Cyber Physical Production*. Cuvillier Verlag, 2017. URL: https://books.google.de/books?id=SfH-DwAAQBAJ.

[96] D. Lacamera. *Embedded Systems Architecture: Explore architectural concepts, pragmatic design patterns, and best practices to produce robust systems*. Packt Publishing, 2018. URL: https://books.google.de/books?id=wnteDwAAQBAJ.

[97] Junghwan Lee and Myungjun Kim. Isolation of shared resources for mixed-criticality autosar applications. *Journal of Computing Science and Engineering*, 16(3):129–142, 2022.

[98] Meir Lehman and Juan Fernandez-Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11:15–44, 01 2001. doi:10.1023/A:1012535017876.

[99] Sebastian Lerch and Jonas Wolf. *Software Update and Upgrade Over the Air (SOTA)*. Vector Informatik GmbH, July 2019. URL: https://cdn.vector.com/cms/content/events/2019/VH/VIC2019/7_Software_Update_and_Upgrade_Over_the_Air__SOTA_.pdf.

[100] Lucid Software Inc. Flowchart symbols and notation, 2023. URL: https://www.lucidchart.com/pages/flowchart-symbols-meaning-explained.

[101] C. Marscholik and P. Subke. *Datenkommunikation im Automobil: Grundlagen, Bussysteme, Protokolle und Anwendungen*. Hüthig Praxis. VDE-Verlag, 2011. URL: https://sisis.rz.htw-berlin.de/inh2011/12393319.pdf.

[102] K. McCord. *Automotive Diagnostic Systems: Understanding OBD I and OBD II*. S-A Design Workbench Series. CarTech, 2011. URL: https://books.google.de/books?id=kyEtsrPk9ZQC.

[103] Tom Mens and Tom Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on Software Engineering*, 30:126 – 139, 03 2004. doi:10.1109/TSE.2004.1265817.

[104] Mercedes-Benz AG. Maximale Flexibilitaet durch das Montagesystem der Zukunft in der Factory 56: Mercedes-Benz stellt Produktion noch flexibler auf, September 2020. URL: https://media.mercedes-benz.com/article/14e4bf4e-1f6e-4e15-acd9-85eb1edb6f18.

[105] Mercedes-Benz AG. Mercedes-Benz Fahrhilfen ab Werk, February 2023. URL: https://www.mercedes-benz.de/passengercars/find-buy/driving-aids/stage.module.html.

[106] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: achievements and challenges. In *Future of Software*

Engineering Proceedings, FOSE 2014, page 70–84, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2593882.2593888.

[107] Microchip. Bootloader Library Help. techreport, Microchip Technology Inc., 2018. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/BootloaderLibrary_v111.pdf.

[108] Seyed Abolfazl Mortazavizadeh, Ahmad Ghaderi, Mohammad Ebrahimi, and Masood Hajian. Recent developments in the vehicle steer-by-wire system. *IEEE Transactions on Transportation Electrification*, 6(3):1226–1235, 2020. doi:10.1109/TTE.2020.3004694.

[109] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In Maurizio Morisio, editor, *Reuse of Off-the-Shelf Components*, pages 287–297, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[110] Varun M. Navale, Kyle Williams, Athanassios Lagospiris, Michael Schaffert, and Markus-Alexander Schweiker. (R)evolution of E/E Architectures. *SAE International Journal of Passenger Cars*, 8(2):282–288, April 2015. doi:10.4271/2015-01-0196.

[111] Nicolas Navet and Françoise Simonot-Lion. *The Automotive Embedded Systems Handbook*. Industrial Information Technology Series. Taylor & Francis / CRC Press, 2008. URL: https://inria.hal.science/inria-00336166.

[112] Tammy Noergaard. Chapter 5 - board memory. In Tammy Noergaard, editor, *Embedded Systems Architecture (Second Edition)*, pages 231–260. Newnes, second edition edition, 2013. URL: https://www.sciencedirect.com/science/article/pii/B9780123821966000054, doi:10.1016/B978-0-12-382196-6.00005-4.

[113] Not A Tesla App. Tesla's Software Update History, February 2023. URL: https://www.notateslaapp.com/software-updates/history/.

[114] Philipp Obergfell. *Entwurfsmethodik für hybride Software- und Systemarchitektur*. phdthesis, Karlsruher Instituts für Technologie, June 2021. URL: https://publikationen.bibliothek.kit.edu/1000134708.

[115] Philipp Obergfell, Florian Oszwald, Matthias Traub, and Eric Sax. Viewpoint-based methodology for adaption of automotive e/e-architectures. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 128–135, 2018. doi:10.1109/ICSA-C.2018.00041.

[116] Mohammed A. Omar. *Final Assembly*, chapter 5, pages 227–248. John Wiley & Sons, Ltd, 2011. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119990888.ch5, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119990888.ch5, doi:10.1002/9781119990888.ch5.

[117] Mohammed A. Omar. *Introduction*, chapter 1, pages 1–14. John Wiley and Sons Ltd, 2011. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119990888.ch1, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119990888.ch1, doi:10.1002/9781119990888.ch1.

[118] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in De-signing Object-Oriented Application Frameworks*. phdthesis, University of Illinois at Urbana-Champaign, 1992.

[119] Keunyoung Park, Sang Guun Yoo, Taejun Kim, and Juho Kim. Jtag security system based on credentials. *J. Electron. Test.*, 26(5):549–557, October 2010. doi:10.1007/s10836-010-5170-y.

[120] Steven Peters, Gisela Lanza, Jun Ni, Xiaoning Jin, and Marcello Colledani. Automotive manufacturing technologies -an international viewpoint. *Manufacturing Review*, 1:1–12, September 2014. doi:10.1051/mfreview/2014010.

[121] PivotPoint Technology Corp. *SysML Specifications: Current Version - OMG SysML 1.6*. PivotPoint Technology Corp., December 2019. URL: https://sysml.org/sysml-specs/.

[122] Priyadarshini, Simon Greiner, Maike Massierer, and Oum-El-Kheir Aktouf. Feature-based software architecture analysis to identify safety and security interactions. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 12–22, 2023. doi:10.1109/ICSA56044.2023.00010.

[123] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

[124] QNX Software Systems Limited. Qnx in automotive - advanced driver assistance systems, 2020. URL: https://www.qnx.com/solutions/industries/automotive/driver_assistance.html.

[125] Konrad Reif. *Automobilelektronik: Eine Einführung für Ingenieure*. Springer Vieweg, 2014.

[126] Shixuan Ren, Zhen Guo, Yuqiao Ning, Qi Yu, and Longhai Yu. In-vehicle network attack based on can and uds: Demonstration and analysis. In *Proceedings of the 2023 5th International Conference on Internet of Things, Automation and Artificial Intelligence*, IoTAAI '23, page 23–29, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3653081.3653086.

[127] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Incorporated, 2015. URL: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/.

[128] Sarah Rogers. How much does an assembly line cost? Technical report, NewStream Enterprises, August 2021. URL: https://www.newstreaming.com/how-much-does-an-assembly-line-cost/.

[129] Marcel Rumez, Daniel Grimm, Reiner Kriesten, and Eric Sax. An overview of automotive service-oriented architectures and implications for security countermeasures. *IEEE Access*, 8:221852–221870, 2020. doi:10.1109/ACCESS.2020.3043070.

[130] Eric Sax, Ralf Reussner, Houssemeddine Guissouma, and Heiko Klare. A survey on the state and future of automotive software release and configuration management. Technical Report 11, Karlsruher Institut für Technologie (KIT), 2017. doi:10.5445/IR/1000075673.

[131] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, Oct 2012. doi:10.1007/s10009-012-0253-y.

[132] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Vieweg+Teubner, Wiesbaden, 2010. URL: https://link.springer.com/book/10.1007/978-3-8348-9368-0, doi:10.1007/978-3-8348-9368-0_5.

[133] Dietmar Scheer, Oliver Glodd, Hartmut Günther, Yves Duhr, and Achim Schmid. STAR3 - Eine neue Generation der E/E-Architektur. In *Sonderprojekte ATZ/MTZ*, volume 25, page 72, 2020. doi:10.1007/s41491-020-0056-5.

[134] Udo Schifferdecker and Christoph Rätz. High-performance computing platforms in the automobile. techreport, Vector Informatik GmbH, 2020. URL: https://cdn.vector.com/cms/content/products/vconnect/docs/2020-02_Automobil-Elektronik_Mastering-Automotive-OTA.pdf.

[135] Marc Schindewolf, Hannes Stoll, Houssem Guissouma, Andreas Puder, Eric Sax, Andreas Vetter, Marcel Rumez, and Jacqueline Henle. A comparison of architecture paradigms for dynamic reconfigurable automotive networks. In *2022 International Conference on Connected Vehicle and Expo (ICCVE)*, pages 1–7, 2022. doi:10.1109/ICCVE52871.2022.9742775.

[136] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, June 2002. doi:10.1145/508448.508472.

[137] Bernhard Schätz, Manfred Broy, Sascha Kirstan, and Helmut Krcmar. What is the benefit of a model-based design of embedded software systems in the car industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 343–369. IGI Global, 01 2011. doi:10.4018/978-1-4666-4301-7.ch017.

[138] Christoph Seidl, David Wille, and Ina Schaefer. *Software Reuse: From Cloned Variants to Managed Software Product Lines*, pages 77–108. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-12157-0_5.

[139] Yehonathan Sharvit. *Data-Oriented Programming*. Manning Publications, New York, 2022. URL: https://www.manning.com/books/data-oriented-programming.

[140] Siemens AG. The True Cost of Downtime 2022. techreport DICS-B10153-00-7600, Siemens AG, 2023. URL: https://assets.new.siemens.com/siemens/assets/api/uuid:3d606495-dbe0-43e4-80b1-d04e27ada920/dics-b10153-00-7600truecostofdowntime2022-144.pdf.

[141] Carsten Sinz. Baubarkeitsprüfung von kraftfahrzeugendurch automatisches beweisen. techreport, Eberhard-Karls-Universität Tübingen, December 1997. URL: https://www.carstensinz.de/theses/Diplomarbeit.pdf.

[142] SofDCar. Software-defined car, 2023. URL: https://sofdcar.de/language/en/.

[143] Sourceability. Q3 2025 electronic components lead time report. Industry report, 2025. Accessed: January 2026. Global delivery lead times for automotive-grade components are cited at approximately 12–14 weeks. URL: https://sourceability.com/post/q3-2025-electronic-components-lead-time-report.

[144] Gabriel Coutinho Sousa Ferreira, Felipe Nunes Gaia, Eduardo Figueiredo, and Marcelo de Almeida Maia. On the use of feature-oriented programming for evolving software product lines — a comparative study. *Science of Computer Programming*, 93:65–85, 2014. Special Issue with Selected Papers from the Brazilian Symposium on Programming Languages (SBLP 2011). URL: https://www.sciencedirect.com/science/article/pii/S0167642313002815, doi:10.1016/j.scico.2013.10.010.

[145] Miroslaw Staron. *Introduction*, pages 1–18. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-65939-4_1.

[146] Statistisches Bundesamt. Key table hourly labour costs in manufacturing, July 2022. URL: https://www.destatis.de/EN/Themes/Countries-Regions/International-Statistics/Data-Topic/Tables/BasicData_LaborCosts.html.

[147] Thilo Streichert and Matthias Traub. *Grundlagen der Elektrik/Elektronik-Architekturen*, pages 15–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-25478-9_2.

[148] Stringo AB. *How customised vehicle movers improve automotive testing efficiency*. Strandvägen 67, 873 71 Nyland, SWEDEN, 2024. URL: https://articles.stringo.com/how-customised-vehicle-movers-improve-automotive-testing-efficiency.

[149] Ingo Stürmer and Hartmut Pohlheim. Model quality assessment in practice: How to measure and assess the quality of software models during the embedded software development process. *Embedded RealTime Software and Systems*, February 2012. URL: https://hal.science/hal-02263433/document.

[150] Sakthivel Manikandan Sundharam, Sebastian Altmeyer, and Nicolas Navet. Model interpretation for an autosar compliant engine control function. In *7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 07 2016.

[151] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[152] TASKING Germany GmbH. *MICROSAR ClassicEmbedded Software für AUTOSAR-Classic-Steuergeräte*, April 2019. URL: https://www.tasking.com/tasking-docs/ctc_user_guide_v6.3r1.pdf.

[153] S. Teuchert and F. Zohm. *Die Elektrik/Elektronik-Architektur der neuen MAN-Truck-Generation*. ATZ Automobiltech Z 122, 2020.

[154] Texas Instruments. Tuning lead angle in drv10x sensorless bldc drivers. techreport, Texas Instruments, October 2020. URL: https://www.ti.com/lit/an/slaa561/slaa561.pdf?ts=1769206145450.

[155] Vivek Thakur. *ASP.NET 3.5 Application Architecture and Design*. Packt Publishing, Birmingham, UK, 2008. URL: https://www.uop.edu.jo/PDF%20File/petra%20university%20ASP.NET_3.5_Application_Architecture_and_Design_October_2008-20612-Part19.pdf.

[156] The MathWorks, Inc. *Generate C Code from Simulink Model*. The MathWorks, Inc., February 2024. URL: https://de.mathworks.com/help/dsp/ug/generate-c-code-from-simulink-model.html.

[157] J.W. Valvano. *Embedded Systems*. Eigenverl. d. Verf., 2012. URL: https://books.google.de/books?id=A4STrgEACAAJ.

[158] VDA e.V. Automotive spice. techreport, German Association of the Automotive Industry e.V., November 2023. URL: https://vda-qmc.de/wp-content/uploads/2023/12/Automotive-SPICE-PAM-v40.pdf.

[159] Vector Informatik GmbH. *DaVinci Developer Classic - Entwurf von AUTOSAR-Classic-Softwarekomponenten*. Vector Informatik GmbH, September 2023. URL: https://www.vector.com/de/de/produkte/produkte-a-z/software/davinci-developer-classic/#c156832.

[160] Vector Informatik GmbH. *TASKING VX-toolset forTriCore User Guide*, January 2024. URL: https://www.vector.com/de/de/produkte/produkte-a-z/embedded-components/microsar/#.

[161] Vector Informatik GmbH. *CANoe - Development and Testing Tool for ECUs and Networks*. Vector Informatik GmbH, Stuttgart, Germany, 2025. Version as of 2025. URL: https://www.vector.com/canoe.

[162] Vector Informatik GmbH. Tools for vehicle diagnostics, software update and ecu flashing, 2025. Accessed: 2025-10-07. URL: https://www.vector.com/int/en/products/application-areas/diagnostics/.

[163] Vector Informatik GmbH. *VN1600 - Netzwerk-Interfaces mit USB und Ethernet für CAN / CAN FD / CAN XL, LIN, K-Line, J1708 und IO*. Vector Informatik GmbH, Ingersheimer Str. 24, 70499 Stuttgart, Germany, 2025. Datenblatt, Benutzerhandbuch und technische Informationen zum VN1610, VN1611, VN1630A, VN1640A, VN1641, VN1670 Interface. URL: https://www.vector.com.

[164] Andreas Vetter, Philipp Obergfell, Houssem Guissouma, Daniel Grimm, Marcel Rumez, and Eric Sax. Development processes in automotive service-oriented architectures. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–7, 2020. doi:10.1109/MECO49872.2020.9134175.

[165] Andreas Florian Vetter. *Hierarchische Versionierung in der Entwicklung von Fahrzeugnetzwerken*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2025. doi:10.5445/IR/1000184383.

[166] Martin Vogel, Peter Knapik, Moritz Cohrs, Bernd Szyperrek, Winfried Püschel, Haiko Etzel, Daniel Fiebig, Andreas Rausch, and Marco Kuhrmann. Metrics in automotive software development: A systematic literature review. *Journal of Software: Evolution and Process*, 33, 05 2020. doi:10.1002/smr.2296.

[167] Vogel Communications Group. Echtzeit: Grundlagen von Echtzeitsystemen, December 2017. URL: https://www.embedded-software-engineering.de/echtzeit-grundlagen-von-echtzeitsystemen-a-5897f8abe8f52370ee04a724b23339f8/.

[168] Andreas Vogelsang. Feature dependencies in automotive software systems: Extent, awareness, and refactoring. *Journal of Systems and Software*, 160:110458, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0164121219302328, doi:10.1016/j.jss.2019.110458.

[169] H.-C. von der Wense and A. J. Pohlmeyer. Building automotive lin applications. In Sven Krueger and Wolfgang Gessner, editors, *Advanced Microsystems for Automotive Applications 2001*, pages 279–292, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[170] Stefan Wagner and Daniel Méndez Fernández. Chapter 3 - analyzing text in software projects. In Christian Bird, Tim Menzies, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 39–72. Morgan Kaufmann, Boston, 2015. doi:10.1016/B978-0-12-411519-4.00003-3.

[171] Julian Weber. *Automotive Development Processes*. Springer Berlin, Heidelberg, 1 edition, November 2014. URL: https://link.springer.com/book/10.1007/978-3-642-01253-2, doi:10.1007/978-3-642-01253-2.

[172] Ellinor Westerberg. Efficient delta based updates for read-only filesystem images: An applied study in how to efficiently update the software of an ecu. mathesis, KTH Royal Institute of Technology, Stockholm, Sweden, March 2021. Master's thesis, Supervisor: Somayeh Aghanavesi; Examiner: Robert Lagerström; Host company: BMW. URL: https://www.diva-portal.org/smash/get/diva2:1538229/FULLTEXT01.pdf.

[173] Tim Wilmshurst. Chapter 17 - more c and the wider c environment. In Tim Wilmshurst, editor, *Designing Embedded Systems with PIC Microcontrollers (Second Edition)*, pages 501–524. Newnes, Boston, second edition edition, 2010. URL: https://www.sciencedirect.com/science/article/pii/B9781856177504100216, doi:10.1016/B978-1-85617-750-4.10021-6.

[174] Fei Xing, Ping Guo, and M.R. Lyu. A novel method for early software quality prediction based on support vector machine. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10 pp.–222, 2005. doi:10.1109/ISSRE.2005.6.

[175] M V Yashina and Ngoulou-A-Ndzeli. Problems of digital diagnostics for prospect of autonomous vehicles implementation. *IOP Conference Series: Materials Science and Engineering*, 832(1):012017, April 2020. URL: https://dx.doi.org/10.1088/1757-899X/832/1/012017, doi:10.1088/1757-899X/832/1/012017.

[176] W. Zimmermann and R. Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur, 5th ed.* ser, ATZ / MTZ-Fachbuch. Wiesbaden, 2014.