Evaluation of SiL Testing Potential—Shifting from HiL by Identifying Compatible Requirements with vECUs

Rudolf Keil 1,2,*; Jan Alexander Tschorn 2; Johannes Tümler 3; and Mehmet Ercan Altinsoy 1

1 Institute of Acoustics and Speech Communication, TUD Dresden University of Technology, Helmholtzstraße 18, 01062 Dresden, Germany; ercan.altinsoy@tu-dresden.de
2 Vehicle Dynamics Department, Volkswagen AG, Berliner Ring 2, 38440 Wolfsburg, Germany; jan.alexander.tschorn@volkswagen.de
3 Anhalt University of Applied Sciences, Bernburger Straße 55, 06366 Köthen, Germany; johannes.tuemler@hs-anhalt.de

* Correspondence: rudolf.keil@tu-dresden.de or rudolf.keil@volkswagen.de

Abstract: Due to the increasing complexity of vehicle software, it is becoming increasingly difficult to comprehensively test all requirements. This inevitably means that alternative test methods, e.g., simulation-based methods, must be used more frequently. However, the challenge involves identifying appropriate requirements that can be technically tested in a simulation environment initially. The present work is aimed at evaluation and optimization of the effectiveness of software-in-the-loop (SiL) simulations in the testing process of vehicle software. The focus is on supporting the testing process by shifting specific test cases from hardware-in-the-loop (HiL) test benches to SiL-based simulations. For this purpose, a systematic approach was developed to analyze and categorize requirements, enabling precise and efficient allocation of test cases. Furthermore, a detailed review and recommendation for improving the ProSTEP iViP standard for virtual electronic control units (vECU) was carried out. The developed matrix associates the defined requirement clusters with different classifications of vECUs, facilitating the identification of suitable test environment types for conducting specific test cases. By assigning test cases to appropriate vECU levels, the testing processes can be targeted and cost-optimized. Finally, the theoretical results were evaluated in an SiL simulation environment. It was observed that a significant part of the requirements could effectively be tested using a vECU. These findings confirmed the potential of SiL simulation environments to not only support, but also enhance, the testing process for vehicle software by providing a cost-effective and flexible complement to traditional HiL test benches.

Keywords: automotive software testing; virtual electronic control units; software-in-the-loop; AUTOSAR standard; validation and verification

1. Introduction

The ongoing digital transformation in the automotive industry necessitates continuous adaptation and development of testing and validation procedures for automotive software. With the introduction of advanced functionalities such as advanced driver assistance systems (ADAS), autonomous driving (AD), and cyber security, the complexity of automotive software is increasing rapidly. Modern vehicles may comprise up to 100 electronic control units (ECUs) and around 100 million lines of code that must work together seamlessly to ensure safe and reliable operation of the vehicle [1]. By 2030, the number of lines of code per vehicle is expected to increase to up to 300 million [2]. As the scope of software expands, the corresponding testing effort increases [3]. Compounding the challenge, there is a trend towards reducing the number of prototypes. This paradigm shift highlights the growing importance of virtual testing methods, positioning them as essential supplements and alternatives to traditional approaches like test driving [4–6]. These methods offer significant advantages, including frontloading, cost efficiency, scalability of test environments, and
faster development cycles [7]. The issue is clearly evident: as the number of functions in cars increases, so does the testing effort. Meanwhile, the number of prototypes continues to be reduced, making virtual testing methods, as mentioned in [6,8], increasingly indispensable. Virtual testing methods are commonly understood to be “X-in-the-Loop” (XiL) systems. These are categorized in the literature as model-in-the-loop (MiL), software-in-the-loop (SiL), hardware-in-the-loop (HiL), and vehicle-in-the-loop (ViL). Figure 1 illustrates that in the early phases of vehicle development, tests are predominantly carried out in a virtual environment. As development progresses, real components are increasingly integrated into the tests. With the test drive phase, the testing takes place entirely in the real environment, leading to an increased test accuracy due to the use of real components [9].

![Figure 1. Overview of vehicle testing methodologies. The illustration has been modified and extended according to [9].](image)

In the survey by Altinger et al. [10], it was reported that approximately 17 to 23% of software tests in the areas of research, pre-development, and series development are already being conducted using SiL. Hardware-specific tests with the whole vehicle, HiL, or PiL still constitute the majority of tests. Since hardware resources are often limited for prototypes, PiL or HiL-specific tests, software testing sometimes needs to be conducted in an MiL or SiL environment, to ensure comprehensive test coverage. However, the implementation of SiL environments depends on overcoming technical limitations, an aspect that is often not mentioned in the existing literature. Furthermore, as indicated by [11], the scope of these sources is limited to functional and behavioral aspects, simulation modeling, and optimal control modeling, which represent only a fraction of the relevant areas in the broad field of software development and testing. As a result, an SiL cannot test all the functionalities that a HiL is capable of assessing. This work aims to bridge this gap by devising a methodological framework to determine the feasibility of implementing functional and non-functional requirements within an SiL environment, thereby optimizing the testing process efficiency.

Virtual ECUs, as a crucial element within the SiL environment, are software-based simulations of physical ECUs used in vehicles. Instead of physical hardware, the functions of the ECUs are simulated in a virtual environment, so that developers and testers can examine the software and its interactions with other systems without having to rely on the target hardware. The ECU software is configured to be cross-compiled on the simulation host (typically an Intel x86 server or desktop) [7]. The ProStep iViP standard classifies vECUs into five different abstraction levels [12]. Figure 2 illustrates the development of vECU abstraction levels in the context of realism and accuracy. It becomes evident that, as the abstraction level increases (from vECU Level 0 to vECU Level 4), the degree of
realism in the simulation also increases. This is visualized by a shift from MiL through SiL to HiL. While vECU Level 0 is primarily located in the MiL domain, Levels 1 to 4 reflect the SiL domain, with a progressive approximation to real hardware conditions in the HiL domain. The end of the spectrum represents the physical ECU—a physical ECU containing the compiled source code original to the target hardware. The deltas ($\Delta 0$ to $\Delta 3$) depicted in Figure 2 represent the discrepancies for individual vECUs, indicating how many requirements and associated test cases cannot be tested compared to a physical ECU due to technical limitations. The higher the level of a vECU, the smaller the delta and, consequently, the more requirements or test cases that can be tested. This difference is based, among other factors, on the extent of unchanged source code, allowing for various levels of realism to be represented in the simulation. The amount of unchanged source code also enhances the credibility of the simulation environment. This is because changes and adaptations of the source code for a different target hardware represent an additional source of errors, which in the worst case may have nothing to do with the original software.

![Figure 2. vECU-Levels and their associated realism from MiL to HiL.](image)

vECUs can be in proprietary formats (e.g., Silver by Synopsys as in [13], VEOS by dSPACE as in [14], or MATLAB/Simulink by MathWorks), each offering specific advantages, depending on the application and system requirements. Determining which test cases can be efficiently and realistically represented in an SiL environment remains a challenging task. In particular, for test cases with specific hardware dependencies, the SiL method has its limitations [15]. This is because testing certain requirements may require specific hardware components that cannot be adequately simulated in a purely virtual environment or require significant effort to do so.

The presented paper introduces an approach to address this challenge by presenting a generic classification of requirements for ECU software, with a specific focus on a brake control system as an example. The aim was to develop a clear decision method that identifies which requirements are suitable for verification in an SiL environment and which are better tested with other methods such as HiL. This systematic approach aims to, not only make the testing process more efficient, but also enhance the quality and reliability of test results. Finally, the developed classification and the resulting action strategy are validated using an example, not only to assess their applicability but also to contribute to the further establishment and optimization of SiL-based testing and validation procedures in the automotive sector.

2. Related Work and Research Questions

In the field of SiL, various papers have been published pursuing various research objectives. A comprehensive overview is provided in the work by Clausen et al. [11]. Their paper discusses, among other topics, the application domains in which SiL environments have been utilized. In a systematic literature review, 17 of the 88 research papers were associated with the automotive domain. Noteworthy is the detailed examination of application areas that have benefited from SiL environments. These studies spanned sub-domains such as ADAS, AD, and control systems for steering and braking, focusing primarily on
functional and behavioral analysis, simulation modeling, control model optimization, and concurrent testing during development. Another significant aspect of the automotive industry, not mentioned in the paper, is the calibration of controller parameters, as detailed in the sources by [13,16].

However, the limitations of SiL discussed in the papers analyzed by Clausen et al. [11] are not exhaustive. Specific limitations mentioned include the development process, aspects of real operational conditions, validation, interoperability, real-time requirements, scalability, and a cost-benefit analysis. In particular, challenges in replicating real operational conditions were noted, requiring clear differentiation depending on the implementation of the SiL environment. In this context, the white paper by ProSTEP iViP [12] offers deeper insights into various approaches specific to the automotive sector that are implementable in an SiL context. Different abstraction levels of vECUs are defined, providing guidelines on how closely the original software is represented. However, the assignment of which tests can be conducted with the various vECU abstraction levels remains an open question. This leads to the following key research questions that this paper aims to answer.

RQ1: What issues are associated with the abstraction levels for virtual ECUs, and how might these be improved? It must be emphasized again whether the existing definitions are sufficient for the use cases, or whether they need to be refined.

RQ2: What software testing requirements can be identified in the automotive sector? To better assign restrictions, it must first be defined which clusters are present in software testing in the automotive sector.

RQ3: How can the abstraction levels of vECUs be meaningfully assigned to different testing areas in the automotive sector? Given that not every SiL offers the same conditions, a gradation must be made to better explore the limits of an SiL.

3. Requirement-Based Testing and Standardization of vECUs

Our methodical approach is based on the following two concepts: requirement-based testing and the associated testing process, as well as the AUTOSAR standard, which also serves as a basis for classifying vECUs. The following Section 3.1 covers the fundamentals of requirement-based testing and discusses the structured process through which software is systematically evaluated and verified. In Section 3.2, the AUTOSAR standard is introduced, which defines the interoperability and modularity of software components. Building on this, Section 3.3 delves into the ProSTEP iViP standard, which defines various vECU levels.

3.1. Requirement-Based Testing in the Automotive Sector

Requirement-based testing is a type of testing that aims to determine the extent to which a system meets specified functional and non-functional requirements. There are different types of software tests, as introduced in [17]. The goal is to verify whether the finished software product complies with and conforms to the specified requirements. This approach is intended to uncover inconsistencies in the software early on, in areas such as performance, reliability, extensibility, user-friendliness, and security [18]. Each requirement should be associated with a test result to ensure traceability. Requirement-based testing, as a testing method, is embedded in a validation process [19].

The validation process aims to systematically detect potential bugs in the software, forming the basis for creating reports on the release and development status, which are essential for the subsequent software release process. This process can be understood as a cycle with three central components (see Figure 3). It is initiated by creating a test plan and specifying the tests. Subsequently, the tests are executed. The results of these tests represent the discrepancies between the desired target state and the actual state. If errors are identified during test specification or test execution, appropriate corrective actions are taken.
Figure 3. Schematic representation of a test process. The illustration has been modified and extended according to [20].

In the test plan, based on the project and feature release plans, the scheduling of individual test executions for the respective features with their abstraction levels takes place. Test executions should be created according to the requirements specified in the feature release plan, including release timeframes. For conducting tests in specific test environment types, a test case description is required. In the case of requirement-based testing, this test case description is based on the underlying requirements and their associated verification criteria. ISO 26262 serves as the authoritative guideline, offering differentiated recommendations for test methods based on the automotive safety integrity level (ASIL) [21].

The specification of test cases should logically follow from the test design, to ensure traceability back to the foundational requirements. The instructions for creating these specifications and implementing the test cases are defined within the test concepts of individual test instances. Different test instances provide various test environments, such as overall test drive, system HiL, ECU HiL, or SiL. The determination of the type of test environment in the verification criteria of the test design is intended to indicate where specific types of tests should logically occur. The responsibility for assessing the test environment lies with the test manager. Ideally, the required properties of the test environment should be defined before provisioning, based on the verification criteria. In the testing process, decisions are made based on the verification criteria, to allocate specific test cases to the corresponding test environments.

The relevance of this scientific research is demonstrated by the necessity for proficiently allocating requirements to an appropriate testing environment. Due to the finite availability of test benches and prototypes for software testing, it is imperative to validate as many requirements as feasible within a scalable testing environment, such as SiL, to maximize efficiency and coverage. An essential point is the variance in the abstraction levels of vECUs, which complicates the specific assignment of test cases. The appropriate assignment of test environments in the verification criteria and the early definition of their properties are important, in order to efficiently utilize the different types of test environments. This presents the challenge of developing a systematic method for evaluating and classifying test environments (in this paper, SiL) that aims to maximize test coverage and make more efficient use of test resources.

3.2. AUTOSAR Classic Platform

The Automotive Open System Architecture (AUTOSAR) is an international standard for software architectures in vehicles. The aim of the standard is to manage the increasing complexity in the development of ECU software. Another key idea is to separate the functional software from the underlying hardware, to facilitate a seamless exchange of software components across different hardware platforms, vehicle models, and even differ-
ent original equipment manufacturers (OEMs) [22]. In the past, software was specifically written for the hardware it would run on [23]. With the AUTOSAR standard, programmers do not need expertise on how the hardware is structured and can generate the source code independently (see Figure 4). By separating the application layer from the underlying hardware architecture, vECUs can be developed, while keeping the functional software unchanged [24].

This standard for software architecture, consisting of a set of specifications, application interfaces, and a methodology, already exists in two variants: AUTOSAR Adaptive and Classic. The AUTOSAR Adaptive Platform, based on a dynamic, POSIX-compliant operating system, is designed for applications with higher computational requirements, such as automated driving and infotainment systems [26]. For a comprehensive understanding, it is noted that this paper exclusively focuses on AUTOSAR Classic, because the standards have different structures and various application areas, necessitating a separate approach.

The architecture of the AUTOSAR Classic Platform (see Figure 5) comprises three software layers running on a microcontroller: the application layer, the runtime environment (RTE), and the basic software (BSW). The application layer, which contains the functional software, is mostly hardware-independent. Communication between software components and access to BSW are facilitated through the RTE, which serves as the complete interface for applications. The individual layers will be described in more detail below [22].

1. Application Layer
The application layer is at the top and includes all parts of the functional software that are mostly hardware-independent. The application layer is divided into individual software components (SWCs), each carrying a functional implementation. According to the AUTOSAR standard, SWCs can be categorized into the following groups [27]:

- Sensor / Actuator Software
Due to the abstraction principle of AUTOSAR, software programmers do not need to specify internal interfaces for connecting to other components or determine which data need to be sent to other ECUs via bus systems when developing an SWC.

2. Runtime Environment

All interfaces of SWCs communicate through the RTE (see Figure 5, in blue), which serves as an implementation of the virtual functional bus (VFB) within an ECU. The RTE acts as a kind of transport layer and forms a virtual communication platform that enables interaction between individual SWCs and the connection to the underlying basic software layer, including the operating system. This allows for abstract and hardware-independent communication. The RTE standardizes communication mechanisms, thus facilitating the portability and reuse of software components across different hardware platforms [28].

3. Basic Software

In the subsequent layer, the BSW is divided into three further sublayers. Directly adjacent to the RTE is the service layer (see Figure 5, in purple). This layer includes, among other things, the operating system and provides modules for memory management, communication control, as well as a range of other services for administration and monitoring. Apart from the operating system, these components are designed to be largely hardware-independent.

Following the service layer is the ECU abstraction layer (see Figure 5, in yellow), which represents the abstracted layer of the driver interface. This segment includes modules for input/output hardware abstraction and interface modules for various bus systems and memory. Specific hardware dependencies begin to emerge at this level, particularly in terms of the selection and configuration of data transmission channels. Complex device drivers (CDD) are a special case that are not standardized. The primary goal of CDDs is to enable the implementation of complex sensors and the control of actuators by directly accessing the microcontroller via specific interrupts or extended microcontroller peripherals or external devices.

The bottom layer of the BSW is the microcontroller abstraction layer (MCAL), which contains internal hardware drivers for internal microcontroller peripherals such as a serial peripheral interface (SPI), watchdog, controller area network (CAN), or analog–digital converter (ADC), etc. The software is directly dependent on the specific hardware implementation, leading to a close integration of software and hardware [22].

Software libraries (e.g., fixed-point mathematical, bit handling, atomic multicore-safe operations, etc.) provide standardized algorithms for basic software modules (BSWM) and SWCs. These include functions in C code that can be in either source or object code and can be used by the BSW, SWCs, the RTE, or complex drivers [29].

In addition to the three software layers (service layer, ECU abstraction layer, and microcontroller abstraction layer), a vertical division into functional subgroups can be made, which will be briefly described below [30].

- **Input/Output (I/O):** Standardized access to sensors, actuators, and ECU peripherals.
- **Memory:** Standardized access to internal and external (non-volatile) memory.
- **Crypto:** Standardized access to cryptographic primitives, including internal and external hardware accelerators.
- **Communication:** Standardized access to in-vehicle network systems, ECU onboard communication systems, and internal ECU software.
• Off-board Communication: Standardized access to Vehicle-to-X communication, wireless network systems in the vehicle, and ECU-external communication systems.

• System: Provision of standardizable services (operating system, timers, fault memory) and ECU-specific services (ECU state management, watchdog manager), as well as library functions.

This concept ultimately serves to abstract the microcontroller’s hardware from the application software. Developers can thus develop their applications independently of the hardware, making the software more portable and reusable. Building on this section, various standards for vECU levels have emerged, which will be discussed in the following section.

3.3. Standardization of vECUs

In general, two different types of vECUs are distinguished: host-compiled vECUs and target hardware-compiled vECUs. Host-compiled vECUs are characterized by the process of cross-compiling ECU software on a simulation host, which typically uses an Intel x86 architecture [7]. This methodology allows for a flexible development environment, but without direct involvement of an ECU hardware model, which requires abstraction of lower, hardware-dependent layers through the simulation of software APIs at defined interfaces in the software stack. The resulting virtual prototype does not represent the complete production code. Certain software layers must be modified or replaced with simulation equivalents to ensure compatibility and functionality within the virtual environment.

In contrast, target hardware-compiled vECUs use compilers tailored to the specific microcontroller or system on chip (SoC). These types of vECUs include detailed hardware simulation models that provide a comprehensive representation of the ECU hardware and peripherals, allowing the execution of production software without any modifications. The software is compiled exactly as intended for the physical ECU in the actual vehicle, requiring extensive and precise modeling of all ECU hardware components.

These two vECU architectures offer their own advantages and use cases, depending on the specific requirements of the development and testing process in the automotive industry. The choice between host and target hardware-compiled vECUs is significantly influenced by factors such as the required accuracy, development speed, available hardware, and specific goals of the software validation. Furthermore, a granular differentiation of vECUs is made with regard to certain scopes of unchanged source code. Table 1 provides an overview of the standard and it is briefly explained below.

Table 1. Abstractions levels of vECU types [31].

<table>
<thead>
<tr>
<th>Level vECU</th>
<th>Application SW</th>
<th>Runtime Environment</th>
<th>Basic Software</th>
<th>OS, Drivers, Complex Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0 vECU</td>
<td>Controller Model</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Level 1 vECU</td>
<td>Production Code</td>
<td>Simulated Code</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Level 2 vECU</td>
<td>Production Code</td>
<td>Production Code</td>
<td>Simulated Code</td>
<td>-</td>
</tr>
<tr>
<td>Level 3 vECU</td>
<td>Production Code</td>
<td>Production Code</td>
<td>Production Code</td>
<td>Simulated Code</td>
</tr>
<tr>
<td>Level 4 vECU</td>
<td>Binary Code</td>
<td>Binary Code</td>
<td>Binary Code</td>
<td>Binary Code</td>
</tr>
</tbody>
</table>

• Level 0 vECU
A Level 0 vECU includes only the behavioral model itself (e.g., as a MATLAB Simulink model) or the C code generated from the behavioral model (e.g., as a functional mock-up unit (FMU) or dynamic link library (DLL)). A Level 0 vECU does not contain production code and is limited to the individual SWCs or functions of the application
layer. The extent of the application layer is irrelevant. Behavioral models focus on testing the functional and logical behavior at an early development stage. For other tests, such as software interface tests, more complex vECU types that are closer to the physical ECU’s production code are required. A Level 0 vECU also falls into the MiL category, because it uses a behavioral model (see Figure 2).

- **Level 1 vECU**
  A Level 1 vECU includes the production code of the application software. The BSW is removed and supplemented with basic runtime and I/O simulation code, to enable the operation of the application software. The simulated code is specifically created or generated for the vECU. In the case of AUTOSAR Classic, this includes the RTE and functionalities that allow the vECU to send and receive data, for example. The vECU can include all parts or only a subset of the application software for a physical ECU.

- **Level 2 vECU**
  Based on a Level 1 vECU, a Level 2 vECU additionally implements simulated BSW functionalities. The basis for the simulated BSW is the production code, which is abstracted and modified at a high level. Unlike Level 1 vECUs, which primarily communicate at the signal level, Level 2 vECUs can enable advanced communication capabilities that operate at both the bus and network levels, such as simulating common vehicle communication protocols like CAN or Ethernet.

- **Level 3 vECU**
  A Level 3 vECU enhances the BSW with the original production code. The essential condition is that both the application and BSW layers are implemented in a hardware-independent manner to ensure smooth integration and functionality within the vECU. In terms of the AUTOSAR standard, hardware dependencies should only occur in the components of the MCAL, the operating system, and certain parts of complex device drivers. Although Level 3 vECUs include a larger coverage of the production code, they can still be complemented with simulated software components, to optimize the completeness and functionality of the vECU. This is relevant for functions that cannot be directly adopted from the production code due to hardware dependencies.

- **Level 4 vECU**
  A Level 4 vECU contains the entire production code compiled directly for the target system, i.e., the physical ECU. It is also possible to include hardware dependencies. This integration allows for the inclusion of all software layers, including the MCAL, the operating system, and complex device drivers, while also considering hardware dependencies. A significant advantage of Level 4 vECUs is that the software uses the same build process for both the simulation and the physical ECU. This eliminates the need for code changes between the vECU and the physical ECU. It also ensures direct transferability of test results and insights between the virtual and real worlds [12].

Synopsys has extended the vECU standard by introducing another sub-level within Level 4 vECU, as depicted in their white paper. This differentiated approach manifests itself in two specific sub-levels (see Table 2). Level 4a and Level 4b, with Level 4b corresponding to ProSTEP iViP’s Level 4 classification. Level 4a represents a new approach, allowing certain software functions to be bypassed, aiming to increase flexibility and simulation speed [7].

<table>
<thead>
<tr>
<th>Level vECU</th>
<th>Application SW</th>
<th>Runtime Environment</th>
<th>Basic Software</th>
<th>OS, Drivers, Complex Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 4a vECU</td>
<td>Binary Code</td>
<td>Binary Code</td>
<td>Binary Code + Bypass</td>
<td>Binary Code + Bypass</td>
</tr>
<tr>
<td>Level 4b vECU</td>
<td>Binary Code</td>
<td>Binary Code</td>
<td>Binary Code</td>
<td>Binary Code</td>
</tr>
</tbody>
</table>
The VDA also addressed the standard in their recommendation VDA 710, but further specified the aspect of signal and message-oriented communication. A signal-oriented implementation facilitates direct communication at the runtime environment (RTE) level. Specifically, data and values are transmitted in a signal-oriented manner through this high-cut interface. In contrast, the low-cut implementation covers the communication layer of the BSW. The signal transmission is message-oriented, characterized by appropriate signal quantization and targeted message packaging within the vECU. For Level 2 vECUs, it is explicitly mentioned that no specific interfaces have been defined as a reference for orientation so far [32].

4. Materials and Methods

The aim of this section is to present a clear and structured process for the analysis and consolidation of requirements and vECU standards in the automotive industry. Figure 6 provides a schematic representation of the applied approach.

Figure 6. Schematic representation of the applied methodology.

For the identification of requirements, an extensive literature review was conducted to consolidate the existing vECU standards and definitions. This review not only considered common standards but also delved into the differences, challenges, and issues arising from various definitions. From the analysis of existing standards and the identified practical problems currently encountered, recommendations for improvement were developed. These recommendations aim at refinement of vECU definitions, to better align them with the practical requirements of the automotive industry.

To answer the first research question, several SiL environments were compared, to identify differences. Various vECUs were utilized in this process. It was examined whether and to what extent SiL environments are used across different departments. To ensure a meaningful analysis, while keeping the effort manageable, four different SiL environments from three distinct domains were investigated, incorporating five different vECUs. Some of the SiL environments from the chassis domain were partially extended and commissioned in-house. A consistent environment, including a residual bus simulation, had to be created to ensure error-free operation of the vECUs. The four SiL environments are summarized in Table 3.
Table 3. Overview of analyzed SiL environments.

<table>
<thead>
<tr>
<th>Environment</th>
<th>vECUs</th>
<th>Tools</th>
<th>Additional Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chassis 1</td>
<td>Electronic Stability Control</td>
<td>MATLAB/Simulink + CarMaker</td>
<td>Brake Hydraulic + Brake Booster</td>
</tr>
<tr>
<td>Chassis 2</td>
<td>Electronic Stability Control</td>
<td>MATLAB/Simulink + CarMaker</td>
<td>Brake Hydraulic + Brake Booster</td>
</tr>
<tr>
<td>Chassis 3</td>
<td>Electronic Stability Control + Zone Controller</td>
<td>Silver + CarMaker</td>
<td>Brake Hydraulic</td>
</tr>
<tr>
<td>Powertrain</td>
<td>Engine Control Module</td>
<td>Silver</td>
<td>Battery Management + Coolant Circuit + Electric Machine</td>
</tr>
</tbody>
</table>

The first two SiL environments have a similar setup and were implemented in a MATLAB/Simulink environment. Since these are two brake control systems, a vehicle dynamics model was also implemented and supplemented with a brake hydraulic model as an s-function. For better integration of the vehicle dynamics model with MATLAB/Simulink, CarMaker for Simulink was used. An electronic stability control (ESC) with a behavioral model of a brake booster was integrated as the vECU.

The third environment also encompassed a brake control system, but it was implemented in Silver. This environment includes an ESC and a zone controller capable of implementing more extensive driving functions. Additionally, a residual bus simulation and brake hydraulic model was included. While the ESC was implemented as an FMU in Silver, a zone controller was specifically created for Silver. This environment was again operated in co-simulation with CarMaker.

The fourth SiL environment was from the powertrain domain and was also implemented in Silver. The engine control module (ECM) was supplemented with an additional behavioral model, such as an electric machine, battery management, or coolant circuit.

For further analysis of the SiL environments, the differences in commissioning and actual handling were examined. Standard functional tests were conducted, and it was also investigated to what extent advanced tests, such as diagnostics, could be performed. Communication with suppliers was established, to understand how the vECUs were created and to what extent they could also be classified by suppliers. The focus of the analysis was always on the properties and limitations of the vECUs.

To obtain a comprehensive and representative set of requirements for physical ECUs, a data analysis of project-specific specification documents and test case catalogs was conducted. By systematically reviewing these documents using specific clustering criteria, relevant requirements could be identified. The specific criteria for clustering were partially based on the number of requirements related to similar properties. Additionally, relevant test cases for specific software releases were considered. This approach aimed to answer the second research question.

Based on the requirements, an assignment to the vECU levels was carried out. Ultimately, an analysis was conducted to determine the technical prerequisites a vECU must meet in order to test a specific requirement cluster. Finally, this methodical approach provides a solid foundation for evaluating and categorizing testing processes using vECU standards in the automotive industry. The results of this method are detailed in Section 5.

5. Results

In this section, the results are summarized, divided into four subsections, each addressing specific aspects of vECUs and the virtual software testing process, along with the associated challenges and solutions.
The first subsection focuses on practical problems with the established ProSTEP iViP standard as they arise in real-world application scenarios. The analysis of discrepancies between theory and practice aimed to shed light on a new approach, including recommendations and modifications, which will be discussed in the second subsection. The proposed adjustments and additions not only serve as suggestions but also form the basis for the final subsection.

The third subsection presents systematically developed clusters of requirements for software validation. These clusters are the result of in-depth analysis, expert interviews, and practical experience. The resulting categorization of requirements aims to assess the technical constraints of virtual methods in the validation process.

The concluding subsection provides a comprehensive overview of which of the defined requirement clusters can be technically tested with different levels of vECUs. It delves into the technical prerequisites that vECUs must meet to validate the categorized requirements. Furthermore, it discusses the remaining restrictions resulting from volatile development methods. This holistic perspective offers a methodical approach to assess the suitability of various vECU configurations for specific requirements and provides valuable insights for the further development and optimization of testing processes in the automotive industry.

5.1. Analysis and Identification of Issues with the Standard for vECUs (RQ1)

The analysis draws on practical experience from day-to-day operations and conversations with suppliers involved in the development of vECUs. Various vECUs were used and analyzed, some created with different development methods and executed in various simulation environments, including proprietary formats or using the FMI standard. This diversity spans across five different suppliers and covers vECUs in areas such as the chassis and drivetrain. An overview of the analyzed vECUs is presented in Table 4. Some suppliers provided their vECUs in two different formats (FMU and DLL). For anonymization purposes, the names of the suppliers were changed.

<table>
<thead>
<tr>
<th>Tier 1</th>
<th>Level vECU ¹</th>
<th>Domain in Vehicle</th>
<th>File Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supplier A</td>
<td>Level 3</td>
<td>Chassis</td>
<td>DLL + FMU</td>
</tr>
<tr>
<td>Supplier B</td>
<td>Level 2–3</td>
<td>Chassis</td>
<td>FMU</td>
</tr>
<tr>
<td>Supplier C</td>
<td>Level 2–3</td>
<td>Chassis</td>
<td>FMU</td>
</tr>
<tr>
<td>Supplier D</td>
<td>Level 3</td>
<td>Powertrain</td>
<td>DLL + FMU</td>
</tr>
<tr>
<td>Supplier E</td>
<td>Level 2</td>
<td>Zone controller</td>
<td>DLL</td>
</tr>
</tbody>
</table>

¹ Assessment from the supplier.

The following subsection will take a closer look at this variance and show how the current definitions could be sharpened to meet the challenges and requirements of this dynamic landscape.

5.1.1. Uncertainties in the Classification of vECUs

While the first two abstraction levels of vECUs are relatively straightforward to interpret, the distinction between Level 2 and Level 3 in the vECU classification is often unclear and can lead to confusion. The uncertainties become apparent in practice, because developers lack clear criteria for when the transition from one level to the next begins. Frequently, this results in a blurry classification, as seen with two of the five suppliers who classified their vECUs as hybrids between vECU Level 2 and 3. This challenge is further exacerbated by the introduction of the new FMI standard 3.0, which is opening up new possibilities for vECUs [33]. vECUs provided as FMUs, due to their lack of message-oriented communication according to the standard, are typically assigned a maximum of vECU
Level 2. However, theoretically, most parts of the software, with few exceptions, could even be presented as production code within an FMU, possibly extending up to the MCAL layer.

The restrictions on signal-oriented communication arise from the technical requirements of the FMI standard. It is, however, possible to implement simulation models as C code for hardware components. Furthermore, within the FMU, signal processing can take place to prepare signals for the communication interface, as schematically illustrated in Figure 7. Within the FMU, signals coming from outside are processed to be prepared for the communication stack as protocol data units (PDU).

![Figure 7. Mechanisms for implementing message-oriented communication within the FMU.](image)

The FMI standard allows for only signal-oriented communication outward, which implies that, in practice, only a high-cut interface can be provided, as mentioned in [32]. This might lead one to believe that implementing the BSW functionalities within this scope is not feasible. However, a solution arises through the implementation of additional software within the FMU, which serves as a supplement to the actual ECU software. This adaptation allows for the augmentation of signals with information and processing them into PDUs. This approach involves packaging data into PDUs, adding information, and scheduling, which is particularly relevant, as bus signals are usually sent cyclically. In a typical scenario, an error management would detect when the prescribed cycle times, for example, of CAN messages, are not adhered to, which should lead to a diagnostic trouble code (DTC). However, this contradicts the continuous reading of inputs within an FMU, which occurs at a specified step size. To bridge this discrepancy, two approaches are possible: either the error management is deactivated or bypassed, or an attempt is made to assign a specific cycle time to each message within the FMU. The second method allows messages to be correctly processed in the lower software layers of the COM stack. This is an important step to ensure the functionality of the FMU in line with existing communication protocols and requirements. Some proprietary software formats offer advanced solutions for simulating the communication interface by integrating the possibility of a virtual CAN bus, enabling message-oriented communication. Two examples of such software solutions are VEOS from dSPACE and Silver from Synopsys. Another interesting approach is that the FMI standard can be used as an interface for vECUs, where the actual vECU is provided in a different proprietary format. This can bypass certain restrictions of the FMI standard, especially in version 2.0.

Interestingly, a supplier classified their vECU as Level 2 due to these communication limitations, even though almost the entire software code was adopted unchanged, which would normally meet the requirements of a Level 3 vECU. Two different factors are being
mixed: the amount of unchanged source code in the vECU, and the additional technical functions provided by the supplier. Important aspects such as providing a virtual CAN bus or access to non-volatile memory (NVM) for configuration also need to be considered.

In summary, it can be noted that the classification of vECUs is complicated by the absence of clear, hard boundaries and definitions. This ambiguity has become evident in practice. In particular, the concepts of high-cut and low-cut interfaces introduce additional uncertainties into the determination and classification of vECUs.

5.1.2. The Variance in the Technical Scope of vECUs

Regardless of the specific implementation of the original source code, it is the technical capabilities and features of a vECU that play a crucial role, both in terms of validating the simulation environment and ensuring testability according to specific requirements. The technical capabilities significantly impact how effectively and comprehensively a vECU can be used to perform the necessary tests and simulations.

In practice, it is evident that different suppliers develop their vECUs differently, leading to a variance in functionality, even with similar vECU classifications. This disparity can arise, among other factors, because the implemented source code does not necessarily correlate with the functionality of the vECU. A prominent example of this is error management, which exists as source code, but the developer does not provide a means to read diagnostic trouble codes (DTCs). These differences in development result in varying capabilities and functions of vECUs, which, in turn, have direct implications for their applicability in different test scenarios. Some examples of technical capabilities that can be considered during the development of vECUs include the following:

1. Communication interface

Common communication interfaces in the automotive sector include CAN, local interconnect network (LIN), Flexray, and Ethernet. When it comes to the communication interface for vECUs, there is a fundamental distinction between signal-oriented and message-oriented communication, each of which has specific application areas and advantages. Signal-oriented communication is often sufficient for component testing, especially when all involved simulation models do not require message-oriented communication within the same environment. In this method, signals are directly injected into the vECU, without more complex communication protocols such as arbitration, checksum calculation, or message counters. They are typically processed immediately at a higher software level. This type of communication is well-suited for virtual test environments, where direct signal transmission is sufficient. The advantage is that, initially, less effort is required to create the vECU. In contrast, message-oriented communication involves the implementation of additional protocol and security aspects, such as identifiers, checksum calculation, and message counters. This type of communication theoretically allows the vECU to communicate with physical ECUs if a compatible hardware interface is available. Often, proprietary formats that provide such virtual communication interfaces are required for implementation. Typically, the hardware components of the communication interface also need to be emulated. A significant advantage of this method is the ability to test specific fault patterns in the bus system, such as incorrect cycle times, timeouts, or errors in checksum calculation. vECUs that support message-oriented communication are particularly well-suited for HiL test benches, where they can easily be used as residual bus simulations, since communication can be processed at the same level as with physical ECUs through hardware interfaces. A prerequisite for this is that they can be simulated in at least real time. This enables realistic simulation and testing of ECUs under various conditions and scenarios.

For completeness, the last possibility is mentioned again, which is typical for Level 4 vECUs, for example, when a hardware emulation of a CAN transceiver is implemented. In this method, communication takes place at the lowest level of the communication protocol, the physical layer. However, the disadvantage is that the
signals exist as physical bits in a virtual simulation environment and are difficult to process further. A direct transmission medium is required. Analogous to the open systems interconnection (OSI) model, communication occurs in the first layer.

2. Memory access
The implementation of non-volatile memory is essential, especially when integrating the BSW of the memory stack. It enables the management and utilization of non-volatile memory within the vECU. Non-volatile memory, typically in the form of an electrically erasable programmable read-only memory (EEPROM), plays a critical role in physical ECUs. It is used for storing essential parameters, calculated values, coding, and diagnostic trouble codes (DTCs), regardless of the power supply status. This type of memory is particularly required for information that extends beyond a single driving cycle, such as stored DTCs that can be read in a workshop or by a dealer. During system startup, in the initialization phase, and during normal operation, this non-volatile memory undergoes several cyclic redundancy checks to ensure the integrity of the stored data and the memory cells themselves. Since it significantly contributes to the storage of relevant parameters and DTCs, simulating this component is of great importance, especially in the context of on-board diagnostics (OBD). In the implementation of non-volatile memory in vECUs, a simple text file is often used to store the data as hexadecimal values. These pieces of information are organized and stored according to memory addresses. This approach facilitates handling and gives the possibility to manually verify or modify data if necessary.

The implementation of non-volatile memory and the ability to modify it form the basis of the configuration and calibration of the vECU. Furthermore, meaningful validation of the vECU can be carried out based on this information by utilizing a dump of an EEPROM from a physical ECU for validation purposes.

3. Calibration
Calibration allows for the adjustment of an ECU’s functions through application parameters, which can include control parameters or characteristic curves, typically stored in non-volatile memory. Especially for virtual tests of the powertrain or chassis, it is essential that the vECU contains the same parameters as those used in the physical vehicle. Various parameters can significantly influence behavior. A typical example of parameter data in the powertrain area, specifically in the ECU, is the engine map. In principle, writing a parameter set is possible by providing non-volatile memory in the form of a text file. However, it would be very cumbersome to save the corresponding parameter datasets as hexadecimal numbers at the correct memory addresses. Therefore, some suppliers offer toolchains that can read in the parameters in the appropriate file formats during the initialization phase. The ability to adjust parameters in a simulation also opens up new possibilities for application in general. In real-world testing, changing and calibrating data is very time-consuming. In contrast, it is conceivable to perform optimization procedures in a simulation where parameters are optimized according to certain criteria. The determined parameters could then be used in the actual vehicle. This would not only save time and costs but also provide the opportunity for more precise and efficient calibrations.

Another important aspect is the adjustment and reading of specific internal signals for certain test cases to monitor the internal states of the ECU during runtime. These internal signals are usually stored in the microprocessor’s RAM and can be read, for example, using the Universal Measurement and Calibration Protocol (XCP). For this purpose, the software supplier must ensure that, for instance, an XCP over Ethernet port is provided. In general, these capabilities go hand in hand with the implementation of the first two points (communication interface and memory access). Nevertheless, it is not a given that these capabilities are actually implemented.
5.2. Approach and Recommendation for Improving the Standard (RQ1)

Given the various special cases observed in practice, a new approach is proposed here to create a better foundation for the final section and the assignment to the requirement clusters. One main issue is that the ProSTEP iViP standard blends the scope of unchanged source code with the technical functions of a vECU, leading to unclear classifications from some suppliers, especially in connection with the FMI standard. Below, the ProSTEP approach to classification is taken up and expanded with information on the technical scope. However, the definitions of the levels should exclusively refer to the scope of unchanged source code and the target hardware for which the source code is compiled. In particular, with regard to the ambiguities between Level 2 and Level 3 vECUs, the following explanation is intended to provide a better overview. As a result, the following recommendations (see Table 5) mainly relate to the share of production code. The percentage breakdown refers to the source code, but simply counting lines of code is not an effective method of evaluation. A clearer method is the categorization into SWCs and BSWMs. The number of SWCs varies depending on the specific project and the scope of implemented functions. In the context of ECUs developed according to the AUTOSAR Classic Platform specification, implemented, for example, with Vector Microsar software, an approximate guideline for the number of BSW modules in such a project can be assumed to be around 40 modules. This value serves as a rough reference for the number of BSWMs that can be expected in a typical project of this kind.

Table 5. Overview of the vECU classification with details of the unmodified source code.

<table>
<thead>
<tr>
<th>Level vECU</th>
<th>Application Software</th>
<th>RTE</th>
<th>Basic Software</th>
<th>OS, Drivers, CDDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>Controller Model</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Level 1</td>
<td>Production Code &gt;1%</td>
<td>Production/ Simulated Code</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Level 2</td>
<td>Production Code 90–100%</td>
<td>Production Code 100%</td>
<td>Production Code &lt; 90%</td>
<td>Simulated Code</td>
</tr>
<tr>
<td>Level 3</td>
<td>Production Code 100%</td>
<td>Production Code 100%</td>
<td>Production Code 90–100%</td>
<td>Simulated Code</td>
</tr>
<tr>
<td>Level 4</td>
<td>Binary Code 100%</td>
<td>Binary Code 100%</td>
<td>Binary Code 100%</td>
<td>Binary Code 100%</td>
</tr>
</tbody>
</table>

- **Level 0 vECU**
  For Level 0 vECUs, everything remains unchanged, so any portion of the source code or SWCs can be implemented as a behavioral model.

- **Level 1 vECU**
  For Level 1 vECUs, there is already production code in the application layer, but this proportion can vary. However, to enable communication between SWCs, the RTE and possibly some portions of the BSW need to be implemented, although these are minimal and not listed in Table 5.

- **Level 2 vECU**
  A Level 2 vECU should primarily consist of unchanged source code in the application layer, with minimal exceptions. Therefore, a benchmark of at least 90% production code is used as a reference. This value is taken as a guideline because it can be easily achieved, and a tolerance range is allowed in case some SWCs are missing. The RTE must be implemented as a complete communication interface. In contrast to the ProSTEP standard, an adjustment is made in the BSW. The majority of BSW can be implemented as simulated source code, but there is the possibility to supplement the BSW with production code for specific functions. It is conceivable that functional blocks in a vertical orientation (see Figure 5), such as the COM or memory stack, could
be realized up to the MCAL. To implement such software components, portions of the hardware peripherals may need to be simulated.

- **Level 3 vECU**
  A Level 3 vECU should primarily consist of production code, with the entire application layer present. All BSWMs that do not require a direct hardware interface should also be in their original source code. However, it has been observed in practice that the implementation of certain modules, such as crypto services, can be associated with significant effort. Therefore, a certain degree of variability is allowed here. None of the five suppliers, for example, integrated the cyber security functionality into the vECU. For Level 3 vECUs, simplified hardware simulations may also be required. Specifically, memory access for the memory stack must be implemented. Since the communication stack is implemented up to the lowest level of the basic software, a low-cut interface should normally be present as well. In this case, the vECU expects PDUs as input data, for example.

- **Level 4 vECU**
  The definition of a Level 4 vECU remains unchanged, as at this level, the entire ECU hardware is emulated. This level of simulation allows for the use of the unmodified binary code of the ECU. While the inclusion of the Synopsys definition is theoretically possible, it represents a proprietary configuration specifically developed for their toolchain and has not seen widespread adoption.

In addition to specifying the unmodified source code, which is represented by the level classification and provides a rough indication of how closely a vECU corresponds to the original software, an extension with technical content will be introduced. This extension encompasses two essential technical areas that include a variety of functions. The problem identified in Section 5.1 is that the quantity of source code does not necessarily correlate with the technical functional scope. Therefore, it is proposed that suppliers should have the option to declare the specific functional scopes using an extended nomenclature. This is particularly relevant for certain test cases that rely on memory access or the communication interface. These functional scopes are mainly relevant for Level 2 and Level 3 vECUs, as they require a certain proportion of basic software functions. The additions will have no impact on Level 0 and Level 1 vECUs, as they do not implement any components of the basic software. Consequently, only Level 2 and Level 3 will be considered further, as they are the most relevant in practice.

To specify the implementation of a communication interface for message-oriented communication, the letter “c” for “Communication” is added to the nomenclature. For example, an implementation that meets the requirements mentioned above would be referred to in the nomenclature as “Level 2c vECU”. The addition of “c” signifies that the vECU has the necessary interface for message-oriented communication at the network level. In Figure 8, the communication stack is shown in more detail to illustrate where the PDUs are externally injected. With a comprehensive implementation of the communication stack, for example, in a CAN protocol, the frames are injected either between the CAN interface and CAN driver or below the CAN driver. In the case of signal-oriented communication, this processing chain is missing, and the signals are injected directly into the RTE. As mentioned above in the FMU example, a vECU that has implemented the communication stack may still only include signal-oriented bus communication.
Due to these requirements, the FMI standard is not suitable for message-oriented communication, because it does not support the corresponding external communication. Instead, a proprietary format is needed, which may also need to simulate parts of the driver or hardware. Such a communication interface is relevant for manipulating the bus system, as it constitutes a significant part of the test cases. Additionally, implementation of the XCP interface should be considered to enable real-time measurement of internal signals. This interface is essential for monitoring the vECU’s internal parameters during operation, allowing for in-depth analysis of system behavior. The second significant technical implementation relates to NVM, which is essential for storing parameter data, encoding, and DTCs. This memory is relevant for numerous technical functions. To symbolize that the relevant aspects of non-volatile memory have been implemented, the nomenclature will be extended with the letter “m” for “Memory”, analogous to the communication interface. The following functions should be ensured by this designation:

- The coding of the vECU is feasible
- Parameters on the vECU can be rewritten
- DTCs can be read

Although the implementation of NVM is already necessary to meet the requirements of a Level 3 vECU, significant challenges exist in making it accessible and usable for users. Without specific knowledge of the storage locations of individual pieces of information within an NVM, a document filled with hexadecimal numbers is hardly usable for the user. Therefore, it is crucial to provide users with simple tools or scripts for accessing and interacting with the memory. Implementation of non-volatile memory could also be considered for a Level 2 vECU, requiring certain parts of the BSW to be implemented. This results in three additional levels, which are briefly described in the following Table 6.
Table 6. Proposed new abstraction levels of vECU types.

<table>
<thead>
<tr>
<th>Level vECU</th>
<th>Non-Volatile Memory</th>
<th>Message-Oriented Communication</th>
<th>Adaptable as a FMU</th>
<th>Adaptable as Proprietary Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Level 2c</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Level 2m</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Level 3</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Level 3c</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>

The additional nomenclature allows for a clear and strict differentiation between the extent of unchanged source code and technical functions. In a Level 2c vECU, the scope of the original source code for the BSW may be limited, but the COM stack can still be implemented for specific use cases. In contrast, a Level 2m vECU could also have certain limitations in production code, preventing it from reaching Level 3. However, the implementation of NVM can be highly relevant for some test cases, which is why this designation is included.

Three out of five suppliers delivered a Level 3 vECU according to their definitions, all of which also implemented an NVM with fault memory, encoding capabilities, and parameterization. Therefore, the scope of a Level 3 vECU should adhere to these criteria by default. However, message-oriented communication is not necessarily guaranteed, even if the original source code is fully implemented up to the MCAL. This is because additional signal processing can occur within an FMU, resulting in the vECU exhibiting signal-oriented communication externally. The drawback is that this additional processing is unrelated to the original software. It is a potential source of errors, which should be considered and potentially validated during vECU development. Due to these restrictions, Level 3c and 2c vECUs cannot be implemented with the FMI standard.

If the proposed approach were applied to the vECUs mentioned at the beginning (Section 5.1), it would provide a more precise classification. The vECUs could be better assigned to a level, and it would be evident which vECUs provide a low-cut interface for communication. According to this approach, the levels would look as shown in Table 7.

Table 7. Overview of analyzed vECUs with new classification.

<table>
<thead>
<tr>
<th>Tier 1</th>
<th>Level vECU 1</th>
<th>Level vECU 2</th>
<th>Domain in Vehicle</th>
<th>File Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supplier A</td>
<td>Level 3</td>
<td>Level 3c</td>
<td>Chassis</td>
<td>DLL, FMU</td>
</tr>
<tr>
<td>Supplier B</td>
<td>Level 2–3</td>
<td>Level 3</td>
<td>Chassis</td>
<td>FMU</td>
</tr>
<tr>
<td>Supplier C</td>
<td>Level 2–3</td>
<td>Level 2m</td>
<td>Chassis</td>
<td>FMU</td>
</tr>
<tr>
<td>Supplier D</td>
<td>Level 3</td>
<td>Level 3c</td>
<td>Powertrain</td>
<td>DLL, FMU</td>
</tr>
<tr>
<td>Supplier E</td>
<td>Level 2</td>
<td>Level 2c</td>
<td>Zone controller</td>
<td>DLL</td>
</tr>
</tbody>
</table>

1 Assessment from the supplier. 2 Assessment with new approach.

The results obtained here provide an initial approach and recommendations for further refining the definition of vECUs. The identified problems and challenges are limited to the analyzed vECUs and reflect the diversity of approaches for creating such vECUs, including highlighting special cases. However, a more precise definition of vECUs can help establish clear requirements and standards for vECUs in the future. This becomes particularly
relevant when an OEM plans to use vECUs for specific test cases as part of software releases. Clear criteria and prerequisites will need to be established for this purpose.

5.3. Requirement Cluster for Software Validation (RQ2)

The release of software for use in a vehicle is based on a software release process that ensures the requirements for the product, especially the software, are met. As explained in Section 5.1, the verification criteria define which requirements are addressed by which types of test environments. This, in turn, depends on which technical functions are implemented in a simulation environment with vECUs and how much original source code is implemented in a vECU. It does not make much sense to use a vECU for test cases that contain a small amount of production code. Therefore, the development of a new method for the testing process using vECUs starts with the creation of requirement clusters. The generation of a cluster takes into account various criteria. A crucial criterion for clustering is the location of the software components to be tested according to the AUTOSAR standard, as shown in the corresponding figure (see Figure 5). This standardization allows for a systematic identification of software components and their functions within the vehicle architecture. A crucial aspect in requirement analysis is the availability of software components. If specific parts of the software are missing, testing of these components is not possible. This insight is fundamental because it directly affects the testability and thus the reliability of the entire testing process. The clustering carried out primarily provides an overview of whether and what fundamental limitations exist regarding the testing of software components. In addition to considering the AUTOSAR architecture, other criteria that are specific to the validation of vECUs are included in the cluster formation. These criteria include, among others, time-critical requirements that must ensure that the vECU reflects valid timing behavior. In relation to control functions, such tasks within the vECU should be properly mapped. Safety requirements, classified according to ASIL, are also relevant. Furthermore, requirements for dynamic functions and logical functions located in the application layer are taken into account in the evaluation. These factors play a significant role in defining the test strategy and help reduce the complexity of the testing processes and increase efficiency.

The resulting requirement clustering is therefore not only a methodological basis for systematic test development but also a strategic tool to ensure the integrity and performance of vehicle software. This systematic approach creates a basis for future test scenarios that can secure both the technical and safety-relevant aspects of ECU software. The basis for the requirement analysis was created through expert interviews, project work experience, and a systematic review of the individual requirement documents from which the requirements arise. Taking the brake control system as an example, this resulted in approximately 18,000 requirements that needed to be tested throughout the project. However, for certain release processes, especially at the beginning of a project, not all of these requirements were used, because the functions were developed in the different steps. The goal of the analysis was to assign these requirements to specific clusters, to better align them with the vECU classifications. The following sections will introduce and describe the identified clusters, which cover nearly all requirements.

1. Flashing
   The commissioning of an ECU is an important step that precedes actual testing. This process involves various steps, including flashing the software, encoding, and writing a dataset. During the flashing process, the actual software is transferred to the flash memory of the ECU’s microcontroller. This requires the presence of a suitable bootloader on the microcontroller. The main goal of this process is to ensure that the software can be flashed onto the hardware without any issues. The most common method is flashing via the on-board diagnostics (OBD) interface, which is standardized in modern vehicles [35].

2. Coding
   Coding an ECU refers to the ability to adapt the software so that it is suitable for a specific vehicle variant and functions according to the specific equipment features.
Typically, the software on an ECU has a wider scope, providing various variants and parameters for different vehicle versions. This, of course, assumes that the corresponding hardware is installed in the vehicle.

A good example of this is the trailer stability function in electronic stability control (ESC), which is available as an optional feature in some vehicle variants when a trailer hitch is present. The function is generally present in the software for all variants but is only activated through appropriate coding. This can also affect the bus communication when additional ECUs are installed in the vehicle. Coding is an essential part of the initial commissioning process, because the software would not function properly without the correct coding. During the testing process, all possible and relevant codings are checked to ensure that they can be coded correctly. At the same time, non-configurable vehicle variants should not be codable. Coding is also done via OBD.

3. Dataset
Another important part of commissioning is the dataset stored in the ECU’s EEPROM. This dataset includes various control parameters for functions and is usually not modified during a test campaign. Changing and writing the dataset is more relevant for the calibrators who tune the vehicles and make individual adjustments. However, a well-tuned dataset still plays a crucial role in evaluating the software. Especially in the chassis area, proper parameterization can have a significant impact on vehicle behavior. Parameterization allows adapting the software’s behavior and performance to specific requirements and conditions, which is essential for assessing and optimizing the software. Writing the datasets can also be performed via OBD.

4. Cyber Security
Cyber security requirements, according to [36], are a response to the increasing potential for cyberattacks on the vehicle network. To ensure data integrity, authenticity, and timeliness, additional security measures are taken. These measures include data source authentication, message integrity verification, and ensuring confidentiality through encryption of relevant data segments. In this context, specific critical CAN messages containing safety-critical signals are encrypted to prevent manipulation from external sources [37]. Security mechanisms must also be thoroughly tested, to ensure their effectiveness in protecting against cyberattacks.

5. Diagnostics
The diagnostics category includes all tests that can be controlled via OBD. Standardization covers not only emission-related electronic modules in a vehicle but rather the entire electronics. This area accounts for a significant portion of test cases and offers a variety of functions that are particularly important for workshops, developers, and maintenance work. The software implements legally required features, including selective monitoring of emission-related systems and monitoring of all major ECUs whose data are accessible through the software. Error messages can be retrieved later through standardized interfaces [35].

The standardized interface used in this context is unified diagnostic services (UDS). UDS is a diagnostic communication protocol used in the field of automotive ECUs and is specified in ISO 14229 [38]. This protocol enables efficient and standardized communication between diagnostic tools and the various ECUs in the vehicle [39].

To perform this function, some components in the basic software (BSW) must be implemented, including the diagnostic communication manager (DCM), which is located in the service layer (see Figure 9).
There are numerous functions that fall under this cluster, and it is not possible to
detail each one individually. However, some of the most important functions include
reading DTCs. OBD defines an extensive list of standardized DTCs, which serve to
identify and diagnose vehicle issues more easily. Additionally, relevant input and
output data can be monitored in real time. OBD provides real-time data on various
vehicle parameters, including engine speed, speed, air temperature, fuel consumption,
and more. The selection of this cluster was based on its relevance to the technical
requirements of a vECU. It constitutes a significant portion of the test cases, which
will be explained in more detail later.

6. Electrical tests
The electrical tests are divided into component tests and software tests. Component
tests are carried out, for example, to determine the durability of components under
certain voltage conditions or short-circuit currents. In contrast, software tests must
ensure that the software responds correctly to specific electrical events such as short
circuits, over-voltage, or under-voltage conditions and detects them. It is checked
whether the corresponding DTCs appear in the fault memory.

7. Interfaces
The requirements for interfaces encompass all physical connections that an ECU has
outside of typical bus communications. These often include sensors and actuators
directly connected to the ECU. During the testing phase, it is ensured that these
interfaces function correctly. An example of this would be the ESC, where wheel
speed sensors play a relevant role. These sensors capture wheel speeds and can
transmit measurement values through their protocol in the case of active wheel speed
sensors. Another example would be the electronic parking brake, which is controlled
by the ESC and serves as an actuator. Finally, human–machine interfaces like control
or display interfaces are also included, often supported by specific protocols. These
specific interfaces must be tested for certain requirements. Due to the variety of
protocols used, the range of requirements for these interfaces can be quite broad.

8. Bus communication
The bus communication cluster encompasses all tests related to the correct imple-
mentation of requirements in the bus system. Various aspects are verified, such as
compliance with prescribed cycle times for message transmission. It ensures that
transmitted values fall within the specified minimum and maximum value ranges.
Furthermore, it checks whether the ECU is capable of detecting errors like incorrect
message counters or errors in the cyclic redundancy check (CRC) calculation. If such
errors occur and a predefined monitoring time has been exceeded, the ECU should
report a corresponding DTC.

9. Network management
The network management cluster primarily focuses on verifying requirements related
to monitoring and controlling the internal states of the ECU. This includes ensuring
that relevant network management messages are transmitted correctly. Furthermore, it checks whether the ECUs enter and wakes up from the sleep mode at the correct times, as specified. A malfunction in this area can lead to the failure to achieve proper bus sleep in the vehicle network. This, in turn, can result in various network participants keeping each other active, leading to unnecessary power consumption and communication issues.

10. Functions

The functions cluster includes all the classic application functions that need to be specifically developed for each ECU and project. While other clusters may contain generic requirements, the requirements in this area are tailored to the specific functionality of the ECU. These functions can be generally categorized into three categories relevant for testing: conditional functions, control functions, and routine functions. Conditional functions can be further divided into if conditions, statement conditions, and logical functions. Conditional functions are expected to implement a specific response based on certain conditions.

Control functions are functions that represent a control loop. They set a reference value and respond to feedback and/or disturbances using a controller. Examples include the lambda control in an internal combustion engine or ESC in dynamic driving situations. Figure 10 illustrates a general control loop.

![Figure 10. Illustration of a simple control loop, modified and extended according to [41].](image)

A routine function, implemented as a function, influences one or more variables as input variables to change other variables as output variables based on the given control arrangement. In this case, the output variable does not affect the input variable, due to the lack of feedback. As an analogy to regulation, consider the example of the accelerator pedal position as an input variable taken by the driver, which represents an open-loop system. Setting a cruise control to maintain a certain speed would be a closed-loop system with feedback and, therefore, a form of regulation.

Conditional functions are also a focus of the validation process. In particular, with conditional functions, a strict approach is applied following the specified specifications, to verify whether certain output variables exist when certain input variables are provided. A simple example of this is the brake disc wiping function in ESC, which requires certain conditions (e.g., activation of the wipers) before it is activated. Subsequently, a cyclic routine is executed, which builds up slight brake pressure in the brake cylinders for a short period to dry the brake discs during wet weather conditions.

Control functions can be evaluated and tested both objectively based on specific evaluation criteria and subjectively. To continue with the cruise control example, a subjective evaluation may involve assessing whether the speed would be adjusted too abruptly through rapid deceleration or acceleration. An objective criterion could be that the control system is stable and can compensate for control deviations without oscillations.

5.4. Mapping of Requirements That Can Be Tested with vECUs (RQ3)

In this section, a systematic overview is provided based on the findings from the previous sections. It outlines the specific requirement areas that can be technically tested...
with different levels of vECUs. This evaluation is particularly relevant for refining the requirements that an OEM should impose on its suppliers, especially when software-related tests are to be conducted in SiL simulation environments. The assessments presented in the matrix are based on experiences gained from projects involving vECUs.

The matrix (Table 8) provides insights into the technical testability of various requirements depending on the level of the vECU. Flashing software onto ECUs requires specific hardware, including at least one microcontroller, to implement the software code. This requirement is only met by a vECU at Level 4. For coding, the use of NVM is necessary. Therefore, this function is only suitable for vECU levels that can simulate NVM, which is available from Level 2. The same requirement applies to parameterization and datasets. The implementation of cyber security measures is associated with significant effort in practice. This effort also depends on whether the security features were implemented in the original system on the software or hardware side. For this reason, such tests should either be conducted on real hardware or with emulated hardware. Diagnosis functions, on the other hand, require NVM and certain parts of the BSW. Furthermore, the implementation of the UDS protocol is relevant for equating the handling of these requirements with a HiL environment. Due to the number of requirements in diagnosis, they should receive special attention. Electrical tests are only possible if the hardware is either emulated or physically present, which is why these requirements strictly do not fall within the scope of vECUs. Interface tests are highly dependent on the hardware. Depending on the implemented hardware components in a Level 3 vECU, some of these tests can be conducted. Bus communication is an important aspect of many test scenarios. Most requirements in this area can be fulfilled as long as a virtual communication interface is available, which applies to all vECU levels with a “c” in the designation. Network management, due to the need to emulate electrical circuits, is also a challenge. However, the states that occur during active operation can be simulated when a communication interface and the corresponding BSW modules are available. The user layer, which includes the logic of the actual software, can be partially tested at a logical level even with a Level 0 vECU, although the complete original source code is usually not available. This also applies to Level 1 vECUs, which often do not contain the entire source code of the user layer. Starting from Level 2, nearly the entire user layer becomes available, allowing for the testing of such requirements. With each increase in the level within the vECU matrix, as you move further to the right, more production code is present. This ensures that the real software exhibits the same behavior and minimizes potential sources of errors that could arise from simulated code and software cuts in the vECU.

The evaluation of a Level 3 vECU for ESC was validated through random testing. Representative test cases from various requirement clusters, usually tested in HiL tests or directly in the vehicle, were used. The test cases were conducted within a simulation environment using the vECU. It was shown that the vECU was capable of covering the majority of test cases. This included testing different coding variants and datasets, which can be easily implemented on the vECU. With current architectures, cybersecurity requirements cannot be met. The necessary components are not integrated into the vECU. In the diagnostic area, there are also some limitations since the UDS protocol is not implemented, resulting in only those diagnostic aspects that the vECU developers have provided interfaces for being testable. Another obstacle arose in testing the hardware interfaces to wheel speed sensors, which were not feasible due to the lack of physical interfaces. In these test cases, the protocol of the wheel speed sensors was checked. However, an additional simulation of the wheel speed sensors would be required. In terms of network management, the vECU was capable of modifying states through specific CAN messages of the network. When testing the application layer, almost all requirements can be covered, but a comprehensive simulation environment, including some track models, is necessary. In summary, it can be stated that approximately 80% of the requirements could be tested using this vECU. The most extensive test scopes were found in the diagnosis, bus communication, and function test areas. These results underscore the efficiency of the vECU in the testing process but
also highlight the limitations imposed by the simulation level and the availability of specific protocol implementations.

Table 8. Overview of which requirements can be technically tested with vECUs.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 2c</th>
<th>Level 2m</th>
<th>Level 3</th>
<th>Level 3c</th>
<th>Level 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flushing</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
</tr>
<tr>
<td>Coding</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dataset</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cyber Security</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
</tr>
<tr>
<td>Diagnostic</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Electrical tests</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
</tr>
<tr>
<td>Interfaces</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bus communication</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Network management</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\* Amount of production code: ✓ Should be completely possible; ◁ Should be partially possible; − Not possible.

6. Discussion of Limitations

In this section, the limitations of this method are highlighted, and suggestions for future approaches are provided.

6.1. AUTOSAR-Specific Limitations

The method presented in this paper is specifically designed for the AUTOSAR Classic Platform. This standard provides a robust framework for the development of embedded software in vehicle ECUs. However, a clear limitation arises for ECUs that do not adhere to the AUTOSAR standard. For such ECUs, the applied method is less suitable, because the specific architecture and software structure of non-AUTOSAR systems cannot be easily assessed using the procedures described here. Therefore, the method cannot be directly applied to the AUTOSAR Adaptive Platform without modifications, as it is based on different principles that enable a more dynamic and flexible software architecture. Within this study, only AUTOSAR-compliant ECUs and hybrid solutions that include both AUTOSAR components and proprietary software elements were considered.

6.2. Variance

Another limiting factor is the variance in the production of vECUs. The results of this study clearly show that there is no uniform approach to creating vECUs. Different manufacturers use different tools and apply varying degrees of resources and efforts to simulate hardware components. As a result, the quality and scope of vECUs vary significantly, directly affecting the amount of original source code that can be implemented and the number of technically testable requirements. The analysis of five different ECUs confirmed that each ECU was developed using different tools and approaches, partially limiting the comparability and transferability of the results.
6.3. Credibility

The final limitation pertains to the validation of the technical feasibility compared to a real system. Empirical investigation into the extent to which vECUs accurately replicate the logic and timing behavior of physical ECUs was not conducted, as demanded by Hansen et al. [42]. Physical ECUs often work with multiple processor cores that process tasks in parallel, handle interrupts, and operate in real time. On the other hand, PCs, which are commonly used for running vECUs, generally execute only a single task at any given moment. Furthermore, microcontrollers often operate in microsecond time frames and control analog signals with response times of less than 1 ms—a scenario that is difficult to simulate with a vECU. These aspects are central to understanding the performance and limitations of vECUs, and they should be more comprehensively examined in future studies, to assess and understand their impact on test results.

7. Conclusions

In the present paper, the growing importance of virtual test methods for software validation in vehicle ECUs has been emphasized. In this context, vECUs are of central importance, but a clear methodology for assessing their suitability for certain software validation tasks is still lacking. In particular, it was unclear which requirements can be tested using vECUs and what limitations may arise. To address this gap, this paper proposed a revision of the existing vECU standard according to ProSTEP iViP and introduced new approaches and additions to enable a better classification and assessment of vECUs.

The study was based on the examination of five different vECUs provided by various suppliers. In parallel, comprehensive analysis and categorization of various requirements were conducted, serving as the basis for requirement-based testing and considering different types of test environments. By using a matrix that assigns various requirement clusters to corresponding vECU abstraction levels, a better allocation of test cases to specific test environment types could be achieved. The results of the study showed that a significant portion of requirements can be virtually implemented through the targeted use of SiL simulation environments. This ultimately reduces costs and time in the testing process.

Finally, an empirical verification was conducted using an SiL simulation environment that implemented a Level 3c vECU for a brake control system. The results of this sample confirmed the findings discussed earlier, indicating that approximately up to 80% of requirements can be successfully tested using such a vECU. This confirmation underscores the practicality of the proposed methodology and demonstrates its potential to enhance the efficiency and effectiveness of virtual testing processes.

In the future, the accuracy of the results obtained in this study could be further enhanced by implementing automation of the simulation environment. Such automation would allow for systematic execution of test cases and objective evaluation of results. This could enable precise quantification of the requirements that can actually be realized in an SiL environment. Additionally, automation provides the opportunity to directly compare results from the SiL environment with results from HiL tests. This comparison is particularly valuable as HiL test benches incorporate real hardware components and the physical interfaces of the vehicle, serving as a good reference for validation. By comparing results from both environments, the validity and limits of SiL-based simulations can be determined even more accurately.

Author Contributions: Conceptualization, R.K.; methodology, R.K.; formal analysis, R.K.; investigation, R.K.; resources, R.K.; data curation, R.K.; writing—original draft preparation, R.K.; writing—review and editing, R.K., J.A.T., J.T. and M.E.A.; visualization, R.K.; supervision, J.A.T., J.T. and M.E.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research project was supported by Volkswagen AG. The results, opinions, and conclusions expressed in this publication are those of the authors and do not necessarily represent the views of Volkswagen AG. The APC was funded by the Open Access Publication Fund of TU Dresden.

Institutional Review Board Statement: Not applicable.
Informed Consent Statement: Not applicable.

Data Availability Statement: In the context of this study, virtual electronic control units and data from suppliers were analyzed, as well as the internal specifications of the original equipment manufacturer. These data are subject to a confidentiality agreement and cannot be provided. The necessary and relevant results are presented in the paper.

Conflicts of Interest: Authors Rudolf Keil and Jan Alexander Tschorn were employed by the company Volkswagen AG. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

ADC Analog Digital Converter  
ASIL Automotive Safety Integrity Levels  
AUTOSAR Automotive Open System Architecture  
BSW Basic Software  
BSWM Basic Software Module  
CAN Controller Area Network  
CDD Complex Device Driver  
COM Communication  
CRC Cyclic Redundancy Check  
DCM Diagnostic Communication Manager  
DLL Dynamic Link Library  
DTC Diagnostic Trouble Code  
EEPROM Electrically Erasable Programmable Read-Only Memory  
ESC Electronic Stability Control  
FMI Functional Mock-up Interface  
FMU Functional Mock-up Unit  
HiL Hardware-in-the-Loop  
LIN Local Interconnect Network  
MCAL Microcontroller Abstraction Layer  
MiL Model-in-the-Loop  
NVM Non-volatile Memory  
OBD On-Board-Diagnose  
OEM Original Equipment Manufacturer  
OSI Open Systems Interconnection  
PDU Protocol Data Unit  
RTE Runtime Environment  
SIL Software-in-the-Loop  
SoC System on Chip  
SPI Serial Peripheral Interface  
SWC Software Component  
UDS Unified Diagnostic Services  
VDA Verband der Automobilindustrie  
vECU virtual Electronic Control Unit  
VFB Virtual Functional Bus  
XCP Universal Measurement and Calibration Protocol

References

4. Abdellatif, H.; Gnadt, C. Use of Simulation for the Homologation of Automated Driving Functions. ATZelectronics Worldw. 2019, 14, 68–71. [CrossRef]

**Vehicles 2024, 6**
32. Verband der Automobilindustrie e. V. VDA 710; Software-in-the-Loop (SiL) Standardisierung. Berlin, Germany, 2022.
36. UNECE, WP. Uniform provisions concerning the approval of vehicles with regards to cyber security and cyber security management system. *Regulation* 2021, 155, 1.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.