

Article

FMU-Based Interaction Layer (FIL) Node for Time-Synchronized Level-4 Multi-vECU Simulation

Harim Lee ¹, Hyeongrae Kim ¹ and Jeonghun Cho ^{2,*}

¹ Mobility Platform Research Center, Korea Electronics Technology Institute, Seongnam-si 13488, Republic of Korea; hrlee@keti.re.kr (H.L.); hrsin09@keti.re.kr (H.K.)

² School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, Republic of Korea

* Correspondence: jcho@knu.ac.kr; Tel.: +82-53-950-5516

Abstract

The increasing complexity of automotive software driven by ADAS and autonomous driving has intensified the need for time-deterministic network validation beyond CAN/LIN, while HIL integration remains constrained by limited ECU prototypes and labor-intensive manual configuration. This paper presents a network-oriented virtual verification environment that couples Renode-based virtual ECUs (vECUs) with FMU-based Interaction Layer (FIL) Nodes automatically generated from DBC specifications. The vECUs provide instruction-accurate execution of unmodified target binaries without physical hardware, while the generated FIL Nodes encapsulate communication behavior as model-based FMUs to maintain continuity from MIL to SIL without manual signal mapping. The proposed framework constructs virtual CAN networks from communication-definition files, reproduces periodic and event-triggered traffic patterns, and supports synchronized multi-ECU co-simulation under master-controlled time stepping. Experimental results show that the vHIL environment reproduces physical ECU timing behavior with a maximum relative error of 0.086–0.171% across all evaluated step sizes (10 μ s to 1000 μ s), confirming binary-level timing fidelity. Replacing vECUs with FIL Nodes for network communication processing significantly reduces wall-clock execution time in multi-node configurations, demonstrating improved scalability without sacrificing determinism. These results demonstrate that the proposed methodology effectively reduces early-stage integration bottlenecks while preserving timing fidelity for automotive networked ECU validation.

Keywords: vECU; vHIL; Renode; AUTOSAR; CAN; IVN



Academic Editors: Calin Iclodean and Dan Moldovanu

Received: 5 June 2026

Revised: 2 July 2026

Accepted: 6 July 2026

Published: 10 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Advanced driver-assistance systems (ADAS) and automated-driving functions are rapidly increasing the number and complexity of software features deployed in modern vehicles. This growth is accelerating the architectural shift from a traditional “one ECU per function” design to integrated platforms built around high-performance computing (HPC) units and zonal controllers, where multiple functions and distributed electronic control units (ECUs) must be validated as an interacting system. As complexity increases, late-stage integration defects become more expensive and more difficult to diagnose, making earlier and more rigorous verification a central requirement for schedule adherence and software quality.

Hardware-in-the-Loop (HIL) simulation is widely adopted to address these needs because it enables closed-loop validation of ECU software while replacing the real plant

with real-time plant models, thereby reducing the dependence on costly prototypes and shortening development iterations [1,2]. However, in practical projects, the expected efficiency gains are often limited by the extensive preparation effort required before tests are executed. This preparation includes plant-model integration, physical I/O configuration, wiring and signal conditioning, communication protocol configuration (e.g., CAN/LIN), and automated test environment construction.

In small enterprises and academic laboratories that frequently handle multi-ECU distributed systems, these steps still rely heavily on manual, project-specific procedures, which are time-consuming and error-prone. Consequently, early integration is repeatedly delayed by configuration failures and rework, forming a dominant schedule bottleneck as illustrated in Figure 1. Recent studies on FMI-based virtual validation, CAN bus co-simulation, and time-synchronized multi-vECU co-simulation further highlight the need for coordinated time management and standardized co-simulation interfaces in such environments [3–7]. These challenges motivate the development of virtual HIL methodologies that shift integration and network validation earlier in the development cycle without sacrificing timing fidelity or reproducibility.

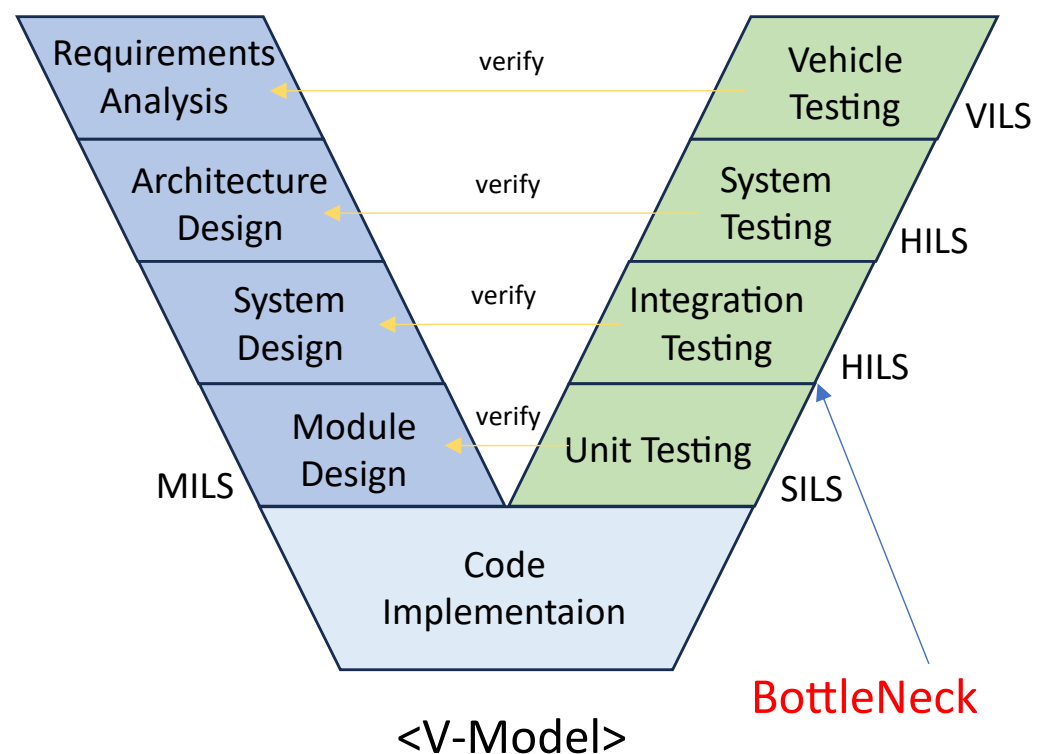


Figure 1. V-model development: traditional process with a bottleneck at integration testing.

Automotive embedded software development commonly follows the V-model, progressing from requirements and design to Model-/Software-/Processor-in-the-Loop (MIL/SIL/PIL) stages and ultimately to Hardware-in-the-Loop (HIL) testing. While project constraints may omit certain stages, preparation effort and integration complexity typically increase toward the right side of the V-model. In particular, HIL testing represents the most preparation-intensive stage, as it simultaneously requires physical ECU availability, deterministic real-time execution, and complete integration of plant models, I/O interfaces, and in-vehicle communication networks [8,9]. Table 1 summarizes the evolution of the Unit Under Test (UUT) and the dominant sources of complexity across XIL levels [10]. While earlier stages primarily focus on model fidelity and software integration, HIL introduces additional challenges related to real-time constraints, hardware dependencies, and

network-level interactions. As a result, delays and failures at this stage often undermine the intended efficiency gains of model-based development.

Table 1. Comparison of XIL test levels and associated complexity factors.

Test Level	UUT (Unit Under Test)	Environment /Inputs	Primary Sources of Complexity Growth
MIL	Control model (SW)	Simulation environment (SW)	Ensuring model fidelity
SIL	Control model (SW)	Simulation environment (SW)	Code generation and integration
PIL	Target ECU's MCU binary	Simulation environment (SW)	Compiler/processor integration; debugging
vHIL	vECU (target binary)	Simulation environment (SW)	Virtual-hardware fidelity; deterministic execution
HIL	Physical ECU (HW/SW)	Real-time simulator (RT HW)	Real-time constraints; physical I/O; network integration

A closer examination of recurring delays in conventional HIL setup reveals three major contributors:

1. Dependence on physical ECU prototypes and bring-up delays.
HIL testing begins only after physical ECU availability, directly coupling verification schedules to prototype delivery. Even after prototype arrival, initialization tasks (boot, memory configuration, flashing, and communication link verification) require substantial manual effort, and hardware-specific issues (e.g., wiring faults or connector problems) further increase delay.
2. Non-standardized interfaces and project-specific custom development.
HIL must integrate analog/digital I/O for sensor/actuator emulation as well as bus interfaces such as CAN and LIN. Conventional solutions often employ custom fault injection, signal conditioning, and interface hardware tailored to each project, which increases setup and maintenance cost while amplifying the probability of configuration errors.
3. High configuration cost for in-vehicle networks and manual mapping.
In multi-ECU systems, network traffic must be reproduced faithfully in the test environment. Message and signal definitions (commonly specified via CAN DBC files) are frequently mapped manually to simulator variables and I/O channels. Minor mistakes (e.g., scaling, endianness, cycle time, or identifier mismatches) can cause system-wide integration failure, leading to long debugging cycles and delaying the first executable test.

These challenges collectively motivate a shift toward configuration-driven and software-defined HIL (SD-HIL) approaches that aim to reduce manual setup effort while preserving deterministic timing behavior [11]. However, a key open question remains: to what extent can network and interface automation be applied without compromising timing fidelity, determinism, and behavioral equivalence to real ECU interactions? Addressing this question is central to enabling earlier, scalable, and reproducible system-level verification in modern automotive development.

To reduce the hardware dependence and move integration earlier in the development cycle, virtual HIL (vHIL) has been proposed as an intermediate validation stage between PIL and physical HIL [12,13]. vHIL executes a virtual ECU (vECU) that executes the same

target binary as the physical ECU, enabling earlier closed-loop testing without requiring physical prototypes.

Prior work by Kim, Kwak, and Cho [12] demonstrated the feasibility of a Level-4 vECU that executes an unmodified AUTOSAR binary with instruction-accurate behavior, establishing a foundation for target-binary verification in virtual environments.

Despite these advances, existing vHIL approaches largely inherit the manual network-configuration workflows of conventional HIL environments: as highlighted in prior FMI-based vECU validation work, users must still manually handle inter-FMU communication and virtual bus configuration [3]. In multi-ECU scenarios, in-vehicle communication networks are still constructed through manual or semi-automated mapping of DBC-defined messages and signals to simulator variables and execution logic.

As a result, network setup remains a dominant source of test readiness delay and configuration errors, limiting the practical scalability and repeatability of vHIL-based validation. This paper addresses this gap by proposing a configuration-driven vHIL architecture that automates network construction and integration for distributed ECU systems. The core idea is to decouple network behavior from individual ECU implementations by introducing FMU-based Interaction Layer nodes (FIL Nodes) that encapsulate communication semantics derived directly from CAN DBC specifications.

These FIL Nodes are automatically generated through a DBC-driven pipeline and coupled with Renode-based vECUs using FMI co-simulation, enabling deterministic, step-synchronized execution of multi-ECU networks without manual signal mapping. This approach differs from commercial DBC-based toolchains such as Vector CANoe or Simulink CAN blocks in three key respects: (1) FIL Nodes are FMI 2.0-compliant FMUs that integrate directly into any FMI-compatible co-simulation master without vendor-specific licenses—in contrast to tools such as the CANoe Interaction Layer, which is OEM-specific; (2) FIL Nodes are generated as role-specific behavioral surrogates for named ECU nodes rather than generic bus-level models, preserving per-node transmit/receive semantics derived from the DBC—not just decoding message payloads as tools such as CANoe or Simulink CAN blocks do, but also encoding the transmission scheduling, event-trigger rules, and ECU-level send/receive roles defined in the DBC; and (3) the pipeline couples FIL Nodes with instruction-accurate Renode vECUs executing unmodified target binaries, enabling a level of binary fidelity not achievable with model-based CAN blocks alone.

By shifting integration and debugging to a pre-hardware stage, vHIL improves not only testing efficiency but also overall quality assurance because early defects can be filtered before scarce physical HIL resources are consumed [14,15]. Figure 2 visualizes this shift-left rationale: the proposed approach removes dominant sources of bring-up delay and reduces late-stage validation rework.

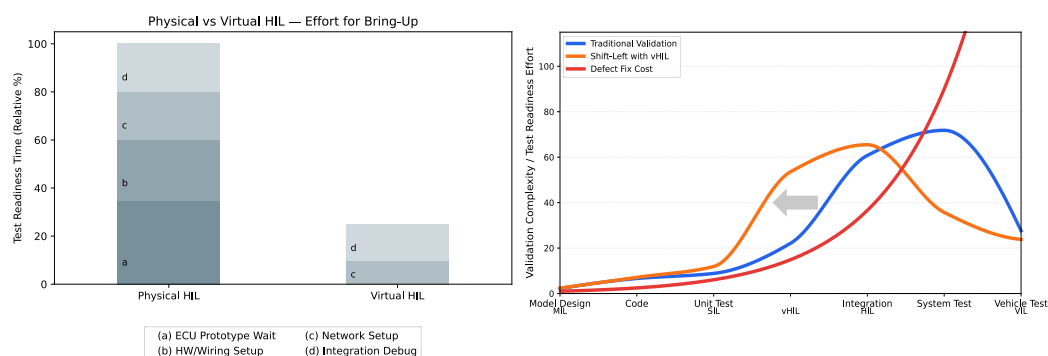


Figure 2. Proposed vHIL concept: (left) removing major HIL preparation bottlenecks by eliminating ECU prototype waiting time and physical HW/wiring setup; (right) enabling a shift-left effect through earlier defect detection and reduced late-stage validation effort.

The main contributions of this work can be summarized as follows:

1. Configuration-driven generation of FMU-based network interaction nodes.
A DBC-driven automation pipeline is introduced to generate FIL Nodes that encapsulate network I/O structures and communication logic as FMI-compliant models, substantially reducing repetitive and error-prone manual mapping tasks in HIL and vHIL setups.
2. Improved repeatability and scalability of multi-ECU vHIL integration.
By automating signal scaling, byte ordering, scheduling attributes, and DBC-to-FMU mapping, the proposed approach minimizes human-induced configuration errors and enables reproducible construction of distributed ECU networks in a virtual environment.
3. Reduction of test readiness effort while preserving timing fidelity.
By eliminating manual DBC-to-simulator signal mapping, the proposed pipeline removes the dominant recurring configuration steps (identifier assignment, scaling, endianness, and cycle-time setup) that otherwise require per-project manual effort. Experimental validation confirms that the resulting vHIL environment maintains timing behavior equivalent to physical ECU measurements, with a maximum relative error of 0.171% across all evaluated step sizes.

The remainder of this paper is organized as follows. Section 2 reviews related work on HIL and vHIL validation, FMI-based co-simulation, and existing DBC-driven network modeling approaches. Section 3 presents the proposed configuration-driven FIL Node methodology, including the modeling of network interaction logic and timing behavior. Section 4 describes the automated generation pipeline and the overall vHIL co-simulation architecture integrating FIL Nodes with Renode-based vECUs. Section 5 reports experimental validation results, focusing on timing fidelity and scalability under different simulation configurations. Finally, Section 6 concludes the paper and discusses limitations and directions for future work.

2. Related Work and Technical Background

2.1. XIL Verification Framework and Virtual ECU Technology

X-in-the-Loop (XIL) verification provides a systematic framework for validating embedded control software by progressively introducing implementation realism through MIL, SIL, PIL, vHIL, and HIL stages [16,17]. While earlier stages focus on functional correctness and code integration, HIL introduces real-time execution, physical I/O, and full network interaction, making it the most resource-intensive phase. Virtual HIL (vHIL) has been proposed to bridge the gap between PIL (or SIL) and physical HIL by replacing physical ECUs with virtual ECUs (vECUs) that execute the same target binary, enabling closed-loop testing without hardware prototypes.

Automotive ECU software commonly follows the AUTOSAR Classic Platform (CP) architecture, which achieves hardware abstraction through a layered structure—Application Layer, Runtime Environment (RTE), and Basic Software (BSW)—enabling portability across physical and virtual targets [18,19]. Recent work demonstrated that unmodified AUTOSAR binaries can be executed with instruction-level accuracy on Renode-based vECUs, forming the basis for binary-faithful vHIL [12]. However, existing vHIL approaches largely inherit the manual network-configuration workflows of conventional HIL setups, leaving multi-ECU communication integration as a persistent bottleneck in scalable and reproducible virtual validation. The proposed FIL Node addresses this gap by introducing a role-specific, auto-generated FMI component that carries the full transmit/receive semantics of a named

ECU node—a structural separation not provided by generic CAN-FMU models or the binary-execution layer of Renode itself.

2.2. FMI-Based Co-Simulation and the Renode Platform

The Functional Mock-up Interface (FMI) provides a standardized, tool-independent mechanism for integrating heterogeneous simulation components [20,21]. FMI 2.0 Co-Simulation (CS) mode, in which each Functional Mock-up Unit (FMU) encapsulates its own solver and a master algorithm coordinates time advancement and data exchange, is well suited to vHIL environments requiring deterministic, step-synchronized execution across vECUs, plant models, and network models. Prior work has applied FMI-based co-simulation to automotive ECU validation [3], real-time co-simulation time management [4], CAN bus integration with Simulink-based vehicle environments [5], and time-synchronized Level-4 multi-vECU simulation [6]; concurrently, the FMI 3.0 Layered Standard for Network Communication (FMI-LS-BUS) has introduced a general-purpose bus abstraction for networked ECU development [7]. Across these works, however, CAN communication semantics are typically modeled at a coarse abstraction level, or require manually constructed components on a per-project basis, limiting scalability and reproducibility in multi-ECU scenarios.

Renode is an open-source virtual development framework that supports execution and debugging of unmodified embedded software across diverse processor architectures [22]. Virtual hardware is described through text-based REPL platform files specifying CPU types, memory maps, and peripheral interfaces, enabling instruction-accurate AUTOSAR binary execution on a host PC. Commercial virtual platform tools such as ARM Fixed Virtual Platforms (FVP) and Fast Models provide instruction-accurate execution for ARM-based targets, but are architecture-specific and require proprietary licenses. Moreover, peripheral modeling in FVP and Fast Models typically relies on vendor-supplied templates with limited customizability, whereas Renode allows user-defined peripheral implementations directly in C# or Python 3.13., making it straightforward to model ECU-specific peripherals such as custom timer and CAN controllers. Renode was therefore selected in this work as an open-source, multi-architecture, and peripherally extensible alternative. In the proposed vHIL architecture, Renode-based vECUs provide binary-level execution fidelity, while FMI-based FIL Nodes supply the explicit interaction layer for network timing and message handling—a separation that is not inherently provided by Renode alone.

Three categories of existing approaches illustrate the gap that motivates the proposed FIL Node. First, although DBC-based toolchains exist for ECU network configuration, the translation of DBC signal definitions into FMI-compatible co-simulation interface structures—including port direction, scaling, and scheduling semantics—has not been automated and remains a manual step requiring expertise in both the DBC format and the FMI standard [3]; as a result, in multi-ECU scenarios, DBC-defined messages and signals must be mapped to simulator variables through manual or semi-automated procedures [5,6]. Second, prior CAN-FMU implementations model the bus as a generic shared medium and do not encode per-node transmit/receive roles or message scheduling semantics derived from the DBC; each ECU node's communication behavior must therefore be specified manually, limiting reusability across different node configurations [5,6]. Third, commercial toolchains such as the Vector CANoe Interaction Layer operate within OEM-specific environments and require vendor licenses, restricting interoperability with open FMI-compatible co-simulation masters; the Simulink Vehicle Network Toolbox CAN blocks additionally rely on host OS-dependent driver interfaces that prevent FMU export—even though Simulink itself supports FMU generation—making the CAN integration layer non-portable across co-simulation environments. Table 2 summarizes these distinctions.

Table 2. Comparison of the proposed FIL Node Auto Generator against existing CAN bus integration toolchains.

Criterion	FIL Node (Proposed)	Vector CANoe IL	Simulink CAN Toolbox
FMI compliance	FMI 2.0 Co-Simulation (full)	FMU import/export within CANoe; IL not exportable as standalone FMU	FMU export incompatible with CAN blocks
Signal mapping	Fully automated from DBC file	Full automation with OEM-supplied DLL; channel-level DBC import only otherwise	Manual per-signal per node
Execution environment	Cross-platform (any FMI 2.0-compatible master)	Windows only; vendor license required	MATLAB R2024b /Simulink required
License model	Open-source; no licensing fee	Commercial (vendor quotation)	Commercial (MathWorks)
ECU role specificity	Role-specific node per named ECU node	Aware via DBC; per-node IL setup requires manual config.	Generic bus-level model

3. Design of the FMU-Based Interaction Layer (FIL) Node

Conventional HIL and vHIL approaches require engineers to manually construct network interaction nodes from DBC files and design specifications, resulting in extensive configuration effort and high susceptibility to parameterization and mapping errors. These limitations are particularly pronounced in distributed multi-ECU scenarios, where network complexity scales rapidly with the number of nodes and messages.

To address this problem, this paper introduces a configuration-driven methodology for constructing CAN-based FMU Interaction Layer (FIL) Nodes. It is important to distinguish the FIL Node from a generic CAN-FMU. A CAN-FMU is any FMU that implements some form of CAN bus communication, typically constructed manually or with tool-specific utilities, and does not carry structural knowledge of message scheduling, signal scaling, or ECU transmit/receive roles. By contrast, an FIL Node is a role-specific FMU generated automatically for a named ECU node: it encodes that node's complete Input–Logic–Output behavior—cyclic transmission schedules, event-trigger rules, signal decapsulation and scaling—derived by static analysis of the DBC. The FIL Node therefore acts as a drop-in behavioral surrogate—a lightweight substitute model that replicates the communication behavior of a physical ECU's network stack far more efficiently than executing the full binary—rather than merely serving as a bus-level communication channel. This distinction is what enables automatic construction of a complete multi-node virtual CAN network from a single DBC file without any manual signal mapping.

The FIL Node generation process is based on static analysis of CAN network definitions and test specifications. A DBC file is parsed to extract message attributes, including transmission modes, periods, trigger conditions, transmitter–receiver relationships, signal grouping, physical units, and scaling parameters. These attributes are systematically mapped to the input and output variables of the FIL Node, allowing the automatic construction of a baseline execution skeleton that reproduces the temporal and logical characteristics of an in-vehicle CAN network without manual signal mapping.

Messages handled by a CAN-based FIL Node are classified according to their transmission semantics. Cyclic messages are processed at fixed periods defined in the DBC and follow a deterministic processing pipeline comprising physical-to-raw scaling, message encapsulation, and insertion into an internal transmission queue. Event-triggered messages are activated when predefined trigger conditions are satisfied, such as internal state changes or the reception of external messages. For these messages, the FIL Node executes a sequence of decapsulation, scaling, internal logic evaluation, re-encapsulation, and queue insertion.

Both cyclic and event-triggered messages generated by the FIL Node are forwarded to the CAN Bus FMU, which loads all received messages into its internal queue and determines transmission order according to CAN arbitration rules (lower message identifier = higher priority). To prevent unrealistically instantaneous message propagation in PC-based simulation environments, the FIL Node applies a deterministic transmission delay derived from message length and bus baud rate. This delay approximates the temporal behavior of a physical CAN bus while remaining compatible with discrete-time FMI-based co-simulation.

To approximate the CAN transmission time in a deterministic manner, we model the frame-level delay as the ideal serialization time over the bus. Let R_b denote the CAN baud rate (bit/s) and D the payload length in bytes ($0 \leq D \leq 8$). For example, using a standard format length without Data field of 44 bits for SOF, arbitration, control, CRC, ACK, and EOF, the transmission delay is as follows:

$$T_{tx}(D) = \frac{44 + 8D}{R_b}. \quad (1)$$

For $R_b = 1$ Mbps and $D = 8$, this yields $T_{tx} = 108 \mu\text{s}$. This model intentionally captures deterministic serialization delay; the scope and rationale of the remaining physical-layer abstractions are discussed in Section Scope and Limitations of the CAN Communication Model.

Figure 3 summarizes the resulting Input–Logic–Output structure of the FIL Node. Incoming messages are decapsulated and scaled at the input stage, processed by internal logic blocks, and re-encapsulated at the output stage before transmission scheduling. This tripartite structure allows test-specific logic to be modified or extended without altering the auto-generated communication skeleton.

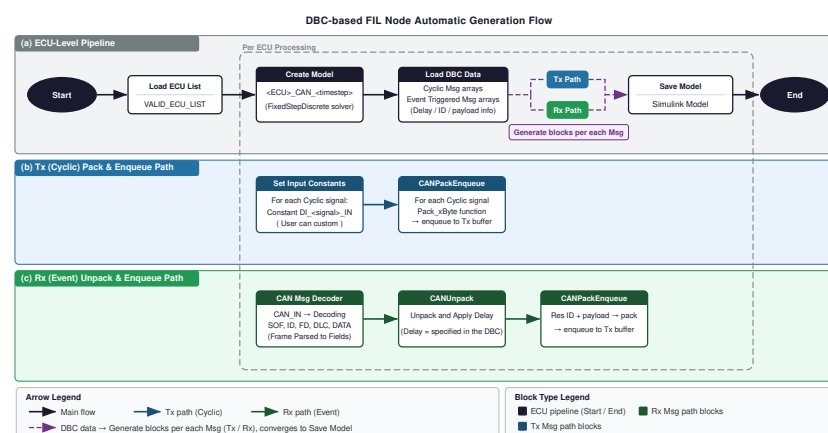


Figure 3. Input– Logic–Output structure of the FIL Node. Incoming messages undergo decapsulation and scaling, followed by internal logic operations, and are re-encapsulated and forwarded to the CAN Bus FMU, which arbitrates transmission order based on message priority.

Overall, the proposed FIL Node design provides a structural advantage over manual and semi-automated CAN modeling approaches by eliminating repetitive configuration tasks and reducing human-induced errors. By separating network interaction behavior into an explicit, auto-generated layer, the method supports scalable integration with automated model and code generation pipelines and enables rapid instantiation of closed-loop vHIL environments directly from DBC-defined communication specifications.

Scope and Limitations of the CAN Communication Model

The FIL Node CAN model targets software-level ECU communication verification under nominal network conditions, and deliberately abstracts several physical-layer behaviors of the CAN bus. Table 3 summarizes each behavior, whether it is captured, and the rationale.

Table 3. Scope of the FIL Node CAN communication model.

CAN Bus Behavior	Modeled	Rationale
Serialization delay T_{tx}	Yes	Computed per Equation (1); deterministic per frame length and baud rate
Message priority (arbitration order)	Partial	CAN Bus FMU loads all messages into its internal queue and applies lower-ID-first arbitration; contention delay omitted; periodic interval unaffected (see text)
Bit stuffing overhead	No	Overhead is data-dependent (runtime payload); exact value not determinable at model generation time; deferred to future work
Bit error / retransmission	No	CRC logic not modeled; targets nominal scenarios; deferred to future work
Host/driver buffering jitter	No	Replaced by FMU step-synchronized deterministic dispatch; see Section 4

Arbitration-induced jitter. In a physical CAN network, a lower-priority message may be deferred by one or more higher-priority transmissions competing in the same arbitration window, causing the frame to depart later than its nominal schedule. However, because automotive CAN traffic is predominantly periodic, this deferral does not alter the inter-message interval of the affected node: if a 100 ms cyclic message is delayed to 105 ms in one cycle, the subsequent transmission occurs at 205 ms, 305 ms, and so forth, preserving the 100 ms period. This behavior was confirmed in the physical ECU measurements conducted for this study, where all cyclic messages maintained stable inter-message intervals despite priority-based arbitration. This characteristic is reproduced in the vHIL environment through the CAN Bus FMU, which loads all messages from connected FIL Nodes into its internal queue and applies CAN arbitration rules, thereby preserving the periodic interval of each node.

Bit stuffing overhead. ISO 11898-1 [23] specifies that after five consecutive bits of identical polarity, a complementary stuff bit is inserted; this rule applies universally to all classical CAN (2.0A/B) implementations regardless of chip vendor. The number of inserted stuff bits depends directly on the runtime payload content—for a standard-format frame with an 8-byte payload, the stuffed portion spans 98 bits and can accommodate at most $\lfloor (98 - 1)/4 \rfloor = 24$ additional bits—and is therefore not determinable at model generation time. Consequently, bit stuffing is not modeled in the FIL Node; precise physical-layer transmission modeling that accounts for data-dependent stuffing overhead is deferred to future work.

Bit error and retransmission. The CAN error confinement and retransmission mechanism relies on CRC generation and hardware-level error signaling performed by the CAN transceiver and controller. These functions are specific to the physical silicon and are not reproduced in the FIL Node. The FIL Node targets nominal communication scenarios in which no transmission errors occur; fault injection and error recovery behavior are deferred to future work, alongside the bit stuffing overhead modeling noted above.

Host/driver buffering jitter. In a physical ECU, CAN frame dispatch is subject to host OS scheduling latency and driver-level buffer management, which introduce non-deterministic jitter into the actual transmission instant. In the FIL Node, this variability is eliminated by design: message transmission is gated exclusively by the FMI step boundary enforced by MasterSim 0.9.6, so a frame scheduled for step k is dispatched at exactly $t_k = k \cdot \Delta t$ without OS-induced delay (Section 4).

4. Auto Generator for FIL Node

4.1. Overview of the Auto Generator Framework

To operationalize the FIL Node design methodology introduced in Section 3, this section presents the Auto Generator: an automated pipeline that transforms CAN network specifications directly into executable FIL Node FMUs. The pipeline ensures that network interaction logic is generated in a consistent, reproducible, and scalable manner—eliminating the manual translation step that remains a primary source of configuration errors in conventional workflows.

As illustrated in Figure 4, the framework takes a CAN DBC file as the primary input and produces FMI 2.0—compliant FIL Node FMUs through a structured, multi-stage process. While DBC files encode essential network information such as message definitions, signal layouts, scaling rules, and ECU relationships, their syntax is optimized for human readability rather than direct structural interpretation. Consequently, a dedicated preprocessing stage is required to enable systematic and unambiguous extraction of network semantics.

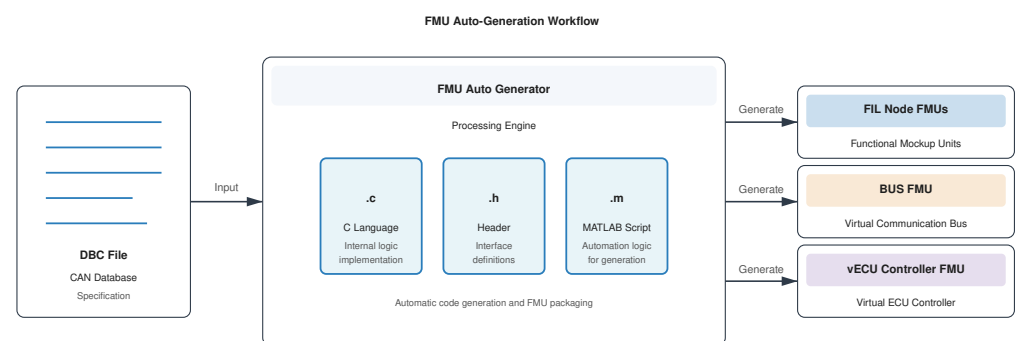


Figure 4. Multi-stage Auto Generator pipeline transforming a DBC file into an FIL Node FMU.

4.2. DBC Preprocessing and Intermediate Representation

The first stage of the framework converts the DBC into a message-oriented intermediate representation expressed in JSON format. In a raw DBC, information is distributed across multiple constructs (e.g., BO, SG, BA_DEF) (Figure 5), which complicates automated analysis and often necessitates proprietary editors. The proposed preprocessing consolidates these scattered elements into a unified, message-centric JSON structure. For each message, the representation captures three categories of information: network topology (transmitter and receiver ECU assignments), protocol parameters (identifier, data length

code, and transmission mode), and per-signal attributes (definitions, physical units, scaling and offset, and endianness).

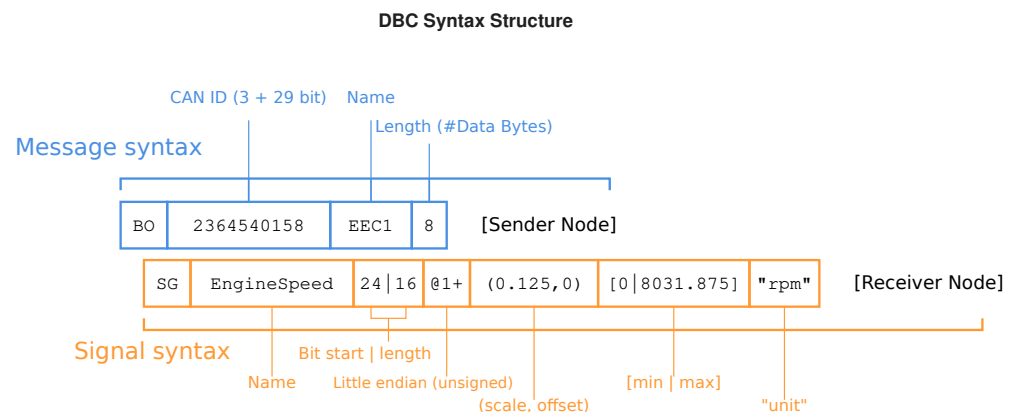


Figure 5. Key structural components of the DBC file used as input to the Auto Generator.

This JSON file is then used as the common input for all subsequent pipeline stages—model skeleton generation, code generation, and FMU export—so that any change to the DBC requires only a re-run of the preprocessing step rather than manual updates across multiple stages.

4.3. Node-Level Structuring and Transmission Semantics

From the intermediate representation, the framework derives node-level specifications by grouping messages according to transmitting and receiving ECUs. Messages are further classified into cyclic and event-triggered categories based on their transmission semantics. Because standard DBC files do not explicitly encode event-trigger conditions, the framework derives a default trigger from the DBC itself. Specifically, the DBC records which ECU transmits each cyclic message; the framework selects the cyclic message of the transmitting ECU associated with the event-triggered message as its default trigger. In the four-ECU experimental scenario, for example, ECU1's event message 0x200 is triggered by the reception of 0x103, which is the cyclic message sent by the ECU listed as the transmitter of 0x103 in the DBC. This default is therefore grounded in the DBC-defined network topology rather than an arbitrary assumption. Once the Simulink model skeleton is generated, users can replace this default trigger logic with application-specific conditions (e.g., a signal threshold or a state-machine transition) by modifying the designated Logic Layer block; the change is then applied to the FMU in the subsequent code-generation and export stages, without altering any other part of the auto-generated skeleton.

When cyclic traffic follows fixed nominal periods—as in the 100 ms scenario evaluated here—the DBC-derived default trigger closely matches physical ECU behavior, and deviating from this assumption introduces negligible behavioral change. For more complex cases involving multi-condition event logic, a real ECU's OS scheduler absorbs any computation-induced response delay without disturbing other cyclic messages; the FIL Node does not capture such computation-induced transmission delays, so they must be explicitly added via the Logic Layer block if behavioral equivalence is required.

The resulting node-level specification defines the complete set of communication responsibilities for each FIL Node and provides the structural foundation for constructing the Input–Logic–Output layers described in Section 3.

4.4. Model Skeleton and Code Generation

Using the node-level specification, the framework generates both a model skeleton and corresponding C/H source code. The model skeleton organizes CAN communication behavior into modular processing stages, including decapsulation with bit-level extraction and scaling, internal logic execution, and encapsulation of outgoing frames for forwarding to the CAN Bus FMU.

DBC-dependent elements—such as message identifiers, payload layouts, scaling rules, and periodic attributes—are generated automatically, while reusable runtime components (e.g., schedulers, encapsulation routines, and FMI interfaces) are incorporated from predefined templates. This separation enables regeneration of FIL Nodes when DBC files change, without duplicating or manually modifying shared logic.

4.5. FMU Export and Deterministic Timing Configuration

In the final stage, the generated model skeleton and source code are packaged as an FMI 2.0 Co-Simulation FMU. This process includes model description generation, compilation, resource organization, and FMU packaging. Key simulation parameters—such as bus baud rate, fixed step size, and transmission-delay settings—are configured during export and embedded into the FMU timing model.

By explicitly parameterizing these timing attributes, the framework ensures deterministic message transmission behavior compatible with master-controlled, step-synchronized co-simulation. Formally, let the discrete simulation time be defined as

$$t_k = k \cdot \Delta t, \quad k = 0, 1, 2, \dots, \quad (2)$$

where Δt is the fixed communication step size. At each step k , MasterSim invokes `fmi2DoStep( $t_k$ ,  $\Delta t$ )` on every FMU in dependency order and does not advance to step $k+1$ until all FMUs have returned successfully—the lock-step invariant

$$t_{\text{FMU}}^{(i)}(k) = t_k \quad \forall i, \quad (3)$$

where $t_{\text{FMU}}^{(i)}(k)$ denotes the simulation time of the i -th FMU at step k . The Renode-based vECU is coupled through a blocking socket handshake: the FIL Node FMU does not return from `fmi2DoStep` until the vECU acknowledges that it has consumed all incoming messages and advanced its internal clock to t_{k+1} . Determinism across the full co-simulation is therefore maintained as follows: (i) MasterSim advances all FMUs—including FIL Nodes and the bus FMU—in lock-step (Equation (3)), so no FMU observes inputs from a future step; (ii) Renode-based vECUs synchronize with MasterSim via a blocking socket handshake at each step boundary, ensuring that the vECU's simulated time does not advance until MasterSim grants permission; (iii) within each step, the FMU execution order is fixed by the MasterSim connection graph, eliminating data-race conditions. It is important to note that this determinism applies to *simulation time*: wall-clock execution time is subject to host OS scheduling and socket latency, which affect only throughput (how long a step takes to complete) and not the logical correctness of the time sequence seen by any FMU. Under normal host conditions (idle system, no resource contention), this OS scheduling and socket overhead manifests only as wall-clock jitter—a run-to-run variation in how long each step takes to execute on the host, not a deviation in the logical simulation time sequence. Because MasterSim's fixed-step master clock is unaffected by wall-clock jitter, the simulated timestamps seen by all FMUs remain identical across runs. Empirically, inter-run wall-clock variation was below 3% across all five repeated runs, confirming that this jitter is negligible in practice.

To ensure repeatability and controlled regeneration, the auto-generation process is organized as a three-stage pipeline, summarized in Algorithm 1. Stage 1 performs DBC-to-JSON conversion, Stage 2 derives node-level workspace structures, and Stage 3 instantiates the model skeleton, generates C/H code, and exports the FMU. Each stage can be executed independently, enabling partial regeneration when network definitions or test requirements change. This structured pipeline supports systematic experimentation and large-scale vHIL deployment, where consistent regeneration of network nodes is essential for scalability and reproducibility.

Algorithm 1 Three-stage build pipeline for FIL Node FMUs.

Procedure MasterPipeline(step, dbcFile, jsonFile, fixedStep_us)

```

1: set_workspace_var("FixedStep_us", fixedStep_us)
2: // STEP = 0: run all stages, 1/2/3: run only that stage
3: if (step == 0 or step == 1) then
4:   // Stage 1: DBC → JSON conversion
5:   step1_dbc_to_json(dbcFile, jsonFile)
6: end if
7: if (step == 0 or step == 2) then
8:   // Stage 2: JSON → workspace arrays
9:   step2_json_to_workspace_can_arrays(jsonFile)
10: end if
11: if (step == 0 or step == 3) then
12:   // Stage 3: workspace → FIL Node model/code & FMU export
13:   step3_workspace_to_FILNode_generate()
14: end if

```

In summary, the proposed auto-generation framework extends configuration-driven FIL Node design into an end-to-end methodology for constructing executable network interaction models. By automating DBC preprocessing, node-level structuring, model and code generation, and FMU export, the framework eliminates manual network configuration bottlenecks while preserving deterministic timing behavior. This approach enables scalable and reproducible construction of multi-ECU vHIL environments directly from communication specifications.

4.6. DBC Complexity and Pipeline Scalability

To characterize the Auto Generator's scalability with respect to DBC complexity, a controlled benchmark was conducted using four synthetic DBC files containing 10, 50, 100, and 500 messages, each with two signals per message (20, 100, 200, and 1000 signals total). Each complexity level was executed three times; Table 4 reports the mean processing time for each pipeline stage.

Table 4. Auto Generator pipeline processing time vs. DBC complexity (mean over 3 runs). Step 1: DBC parsing; Step 2: workspace construction; Step 3: Simulink model generation; Step 4: FMU export (C code generation, compilation, packaging).

Messages	Signals	Step 1 (s)	Step 2 (s)	Step 3 (s)	Step 4 (s)	Total (s)
10	20	0.31	0.09	22.6	144.0	167.1
50	100	0.21	0.05	11.3	134.8	146.3
100	200	0.36	0.10	15.6	143.1	159.2
500	1000	1.91	0.47	57.5	258.2	318.1

FMU export (Step 4) dominates total generation time, accounting for 85–95% across all complexity levels. Steps 1 and 2 are negligible even at 500 messages (1.91 s and 0.47 s respec-

tively). Step 3 scales with the number of Simulink blocks; the apparent non-monotonicity between 10 and 50 messages reflects Simulink engine cold-start overhead on the first `new_system` call within a session, after which Step 3 scales linearly with message count. FMU export exhibits a near-constant plateau of approximately 140 s for message counts up to 100, reflecting the fixed compilation cost of shared C source files (`Pack_ver2.c`, `can_buf.c`, etc.) that are independent of DBC content. At 500 messages, the generated C source grows sufficiently to increase FMU export time to 258 s. Within the practically relevant range for single-segment automotive DBC files (typically 10–100 messages per node), total Auto Generator runtime remains below 170 s, confirming that the pipeline is suitable for iterative use during development cycles.

For the 4-ECU project DBC used in the main experimental validation (12 messages total), the Auto Generator completed the full pipeline in 66.95 s (Step 1: 1.21 s; Step 2: 0.56 s; Step 3: 65.18 s), generating four FIL Node FMUs with zero configuration errors.

5. Experimental Validation

5.1. Validation Objectives and Experimental Scope

This section validates the proposed configuration-driven vHIL environment with respect to two complementary aspects: (i) timing fidelity of CAN communication relative to physical ECU measurements, and (ii) scalability of the co-simulation under varying node compositions and simulation step sizes. The experiments are designed to assess whether automatically generated FIL Nodes can faithfully reproduce cyclic and event-triggered CAN traffic while enabling scalable multi-node simulation in a virtual environment.

An experimental matrix comprising 28 configurations was constructed by varying two factors: simulation step size (7 levels: 10, 20, 50, 100, 200, 500, and 1000 μ s) and node composition within a fixed four-node network (vECU : FIL Node = 4:0, 3:1, 2:2, 1:3). To improve robustness, each configuration was executed five times, and both aggregated 3D views and factor-isolated 2D analyses are reported to clarify the individual influence of step size and node composition.

5.2. Co-Simulation Environment and Reference Setup

The vHIL experiments were conducted using an FMI-based co-simulation environment orchestrated by MasterSim, which coordinates time advancement and data exchange among Renode-based vECUs, auto-generated FIL Node FMUs, and a CAN bus FMU (Figure 6). FIL Nodes communicate with vECU controllers through the bus FMU, while each vECU executes the target ECU binary on the Renode platform.

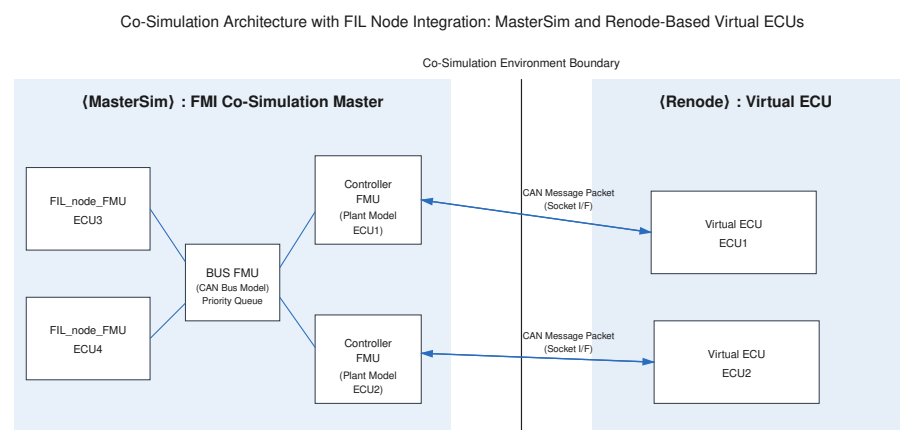


Figure 6. Co-simulation architecture integrating auto-generated FIL Nodes with MasterSim and Renode-based Virtual ECUs.

To establish reference data for timing fidelity, an equivalent network configuration was implemented on physical ECUs, as shown in Figure 7. In this setup, the same target binary was executed on real ECU hardware, and CANoe records the CAN traffic to serve as a reference for evaluating the vHIL environment. These measurements serve as the baseline against which the vHIL results are compared.

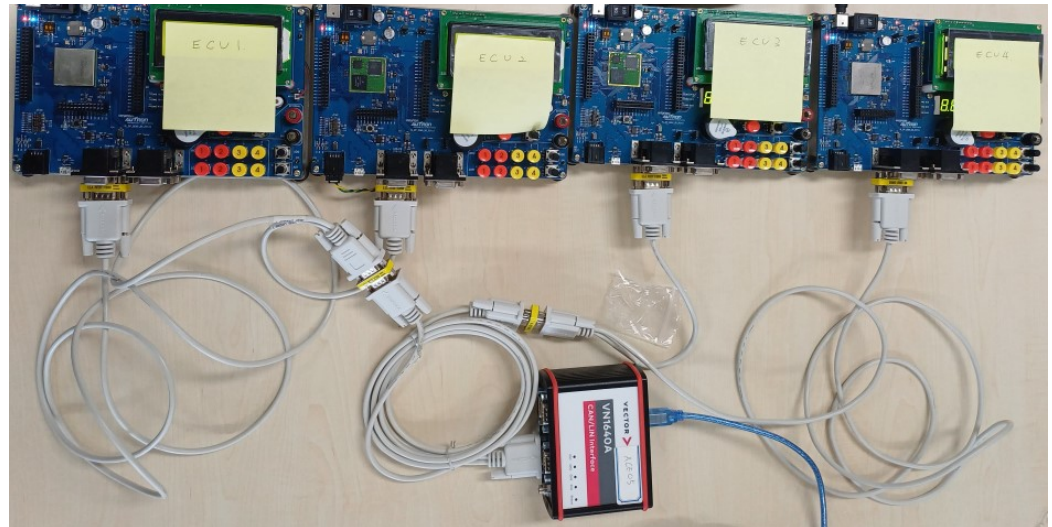


Figure 7. Physical ECU-based measurement setup. The target binary is executed on the real ECU, and CANoe records the CAN traffic to serve as a reference for evaluating the vHIL environment.

5.3. Network Configuration and Test Scenario

The experimental CAN network consists of four ECUs (ECU1–ECU4) that exchange both cyclic and event-triggered messages. Each ECU transmits periodic messages with a nominal cycle time of 100 ms; the payload of each cyclic message includes a counter signal that increments by one per period, yielding a predictable but non-static frame sequence across the simulation. Upon reception of a periodic message, a corresponding event-triggered message is transmitted, forming a closed-loop interaction pattern representative of distributed ECU coordination. The co-simulation boundary interfaces among MasterSim, FIL Node FMU, CAN Bus FMU, vECU FMU controller, and Renode are summarized in Table 5.

Table 5. Interface definition for the co-simulation boundary among MasterSim, FIL Node FMU, CAN Bus FMU, vECU FMU controller, and Renode. SOF: Start of Frame; FD: CAN FD flag (1 = CAN FD frame, 0 = classical CAN); frame type (standard/extended) is determined by the ID field ($ID \leq 0x7FF$: standard 11-bit; $ID > 0x7FF$: extended 29-bit); DLC: Data Length Code; B0–B7: payload bytes separated by ‘:’.

Interface & Component Pair	Protocol	Type	Data/Format
Time coordination (MasterSim → FMU)	FMI fmi2DoStep	—	Simulation time $t_k = k \cdot \Delta t$; step size Δt
CAN frame exchange (FIL Node FMU ↔ CAN Bus FMU)	FMI String	String	"SOF-ID-FD-DLC-PAYLOAD"; parsed inside FMU
CAN frame exchange (CAN Bus FMU ↔ vECU FMU Controller)	FMI String	String	"SOF-ID-FD-DLC-PAYLOAD"; parsed inside FMU
Socket synchronization (vECU FMU ctrl ↔ Renode)	Socket (blocking)	String	"SOF-ID-FD-DLC-PAYLOAD"; + step sync ACK

Table 6 summarizes the assigned cyclic and event-triggered message identifiers for each ECU. This configuration allows validation of both time-triggered and event-driven communication semantics generated automatically from the DBC file, ensuring consistency with the behavior observed on the physical ECU network.

Table 6. Cyclic and event messages assigned to each ECU.

	ECU1	ECU2	ECU3	ECU4
Cyclic Messages _{100ms}	0x100	0x103	0x104	0x105
	0x101			
	0x102			
Event Messages _{triggered Id}	0x200 _{0x103}	0x203 _{0x100}	0x204 _{0x101}	0x205 _{0x102}
	0x201 _{0x104}			
	0x202 _{0x105}			

5.4. Measurement Procedure and Reproducibility

For each configuration in the experimental matrix, CAN messages were logged for a duration of 5 s. To account for run-to-run variability, each experiment was repeated five times. The recorded logs include message identifiers, timestamps, and inter-arrival intervals, enabling detailed analysis of timing behavior.

In the vHIL environment, timestamps are derived from the master-controlled simulation clock (MasterSim fixed-step master), ensuring fully deterministic time progression independent of host OS scheduling. In the physical ECU setup, timestamps recorded by Vector CANoe (hardware timestamp resolution: 1 µs) are used as the reference. Reported results are presented as averages across the five repeated runs; run-to-run variation is discussed in the results.

The software environment used in this study is summarized in Table 7.

Table 7. Software and hardware environment for experimental validation.

Component	Version / Specification
Co-simulation master	MasterSim 0.9.6 (fixed-step FMI 2.0 CS)
vECU platform	Renode 1.14.0
FIL Node toolchain	MATLAB/Simulink R2023b + FMI Kit
Physical reference	Vector CANoe 16.0, hardware timestamp
Host OS	Windows 10 64-bit
Host CPU	Intel Core i7-12700, 32 GB RAM

5.5. Evaluation Metrics

Timing fidelity is evaluated by comparing the inter-message intervals and transmission patterns observed in the vHIL environment with those measured on physical ECUs. The primary metric is the relative error between the average message period in vHIL and the physical reference.

Scalability is evaluated using wall-clock simulation time as a function of (i) the number of vECUs, (ii) the number of FIL Nodes, and (iii) the simulation step size. This metric captures the computational overhead introduced by different node compositions and highlights the performance impact of replacing vECUs with FIL Nodes in large-scale simulations.

In summary, the experimental setup combines physical ECU measurements with controlled vHIL simulations to evaluate both timing fidelity and scalability. By systematically varying simulation parameters and repeating measurements, the experiments provide a reproducible basis for assessing the effectiveness of the proposed FIL Node-based vHIL framework.

5.6. Fidelity Results

To assess timing fidelity, the same ECU binary was deployed on a Renode-based vECU and its CAN message transmission period was compared against a physical ECU measurement. The target binary is configured with a nominal cycle time of 100 ms; the physical ECU measured by CANoe yielded a steady-state interval of 100.010 ms, which serves as the sole reference data throughout the comparison.

Table 8 summarizes the results across seven simulation step sizes (10–1000 μ s), each repeated five times. Two trends are evident. First, the relative error between the vHIL message interval and the physical reference remains below 0.2% across all configurations, ranging from 0.086% at 10 μ s to 0.171% at 100 μ s. Second, the error stabilizes beyond 100 μ s (0.171% at both 100 μ s and 200 μ s), indicating that the step-size-induced timing quantization error saturates at this point and does not worsen with further increases in step size. In addition, run-to-run variation was less than 0.01 ms across all configurations, confirming that the fixed-step deterministic execution produces consistent results regardless of when the simulation is run.

Table 8. Average CAN message interval, relative error, and message-interval jitter across simulation step sizes compared with the physical ECU reference. Reference data: physical ECU measurement (100.010 ms). Reported values are averages over five repeated runs. Jitter is the standard deviation of individual inter-message intervals within a single run.

Step Size	Avg. Message Interval	Relative Error	Jitter (Std)
Physical ECU (Reference)	100.010 ms	—	—
10 μ s	100.096 ms	0.086%	18 μ s
20 μ s	100.138 ms	0.128%	15 μ s
50 μ s	100.167 ms	0.157%	24 μ s
100 μ s	100.181 ms	0.171%	45 μ s
200 μ s	100.181 ms	0.171%	97 μ s
500 μ s	100.177 ms	0.167%	217 μ s
1000 μ s	100.177 ms	0.167%	344 μ s

These results demonstrate that the Renode-based vECU replicates binary-level ECU timing behavior with high fidelity across the full range of evaluated step sizes. This capability enables earlier defect detection in the virtual environment, without requiring physical hardware prototypes, directly supporting shift-left verification strategies.

To clarify the origin of the observed 0.086–0.171% error, we decompose it into four candidate sources. First, MasterSim scheduling quantization: The master advances simulation time in fixed increments of Δt , and a quantization error arises only when the message period is not an exact integer multiple of Δt . All step sizes evaluated here (10–1000 μ s) divide evenly into the 100 ms period, so this source contributes exactly zero to the observed error. Second, IPC socket latency: The blocking socket handshake between the vECU FMU controller and Renode introduces per-step OS-level delay. However, the FMI lock-step mechanism ties simulation-time advance to the completion of `fmi2DoStep`, not to wall-clock elapsed time; socket latency therefore affects only execution throughput, not simulation-time accuracy. The run-to-run variation below 0.01 ms across all configurations confirms this: OS-level socket noise does not propagate into simulation timing. Third, bit stuffing overhead: As discussed in Section Scope and Limitations of the CAN Communication Model, ISO 11898-1 mandates insertion of up to $\lfloor (98 - 1)/4 \rfloor = 24$ stuff bits for a standard-format frame with an 8-byte payload, where the stuffed portion spans

98 bits. At a CAN baud rate of 500 kbit/s, this corresponds to a maximum additional frame transmission time of $24 \times 2 \mu\text{s} = 48 \mu\text{s}$, giving an upper-bound relative contribution of $48 \mu\text{s}/100 \text{ ms} = 0.048\%$ to the message period. Because the FIL Node omits bit stuffing (Section Scope and Limitations of the CAN Communication Model), this overhead is present in the physical ECU reference but absent from the vHIL measurement. However, in the specific test scenario used here, the transmitted payload contains a signal that increments by one each period; the resulting frame-to-frame bit-pattern change is minimal (a unit increment typically alters only the least significant bits, changing the stuff bit count by at most one to two bits per signal), so the stuff bit overhead remains approximately constant across frames. This near-constant overhead is therefore substantially absorbed into the physical reference measurement (100.010 ms) and contributes negligible variable error to Table 8. For signals with large or unpredictable value changes, stuff bit count would vary more significantly frame-to-frame, introducing additional period jitter not captured by the FIL Node model; this remains a limitation identified in Section Scope and Limitations of the CAN Communication Model. Fourth, and dominantly, Renode internal timer discretization: Renode's virtual timer advances in units of Δt per FMI step signal and dispatches the CAN message when it crosses the 100 ms threshold. At coarser step sizes the timer crosses this threshold later by up to one full increment, introducing a positive period offset that grows with Δt . The stabilization at 0.171% for step sizes of 100 μs and above indicates that the rounding is bounded by a fixed resolution floor rather than growing indefinitely. Since the first two sources contribute zero and the third is approximately constant and substantially absorbed into the reference for this scenario, the entire 0.086%–0.171% range is attributable to Renode internal timer discretization alone.

5.7. Scalability Results

Wall-clock execution time was measured for a 5-second virtual-time simulation across all 28 configurations. Figure 8 presents a 3D surface plot of these results over the full step-size and node-composition space.

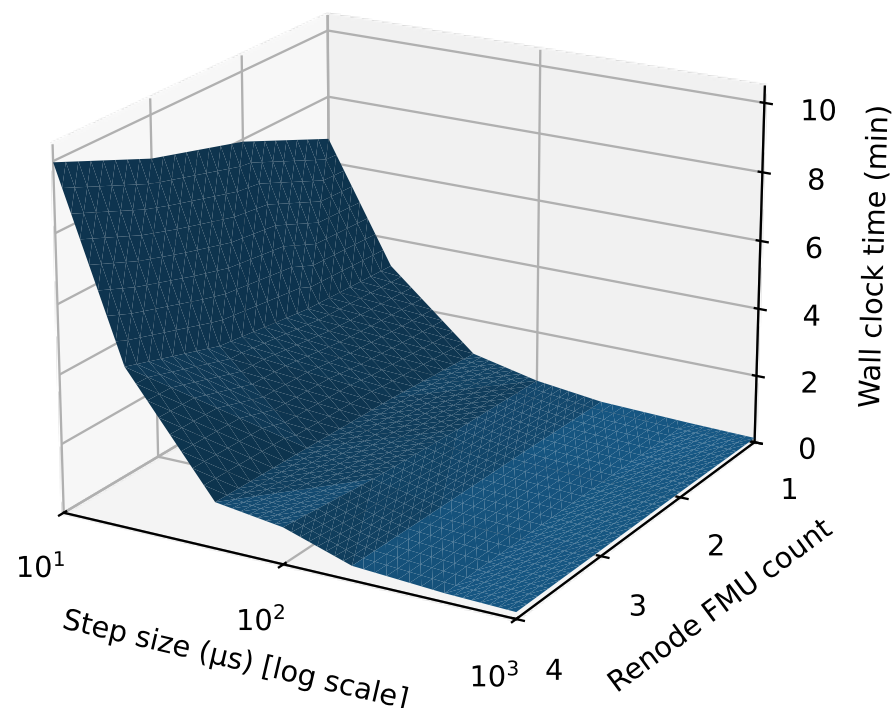


Figure 8. 3D surface plot of wall-clock execution time for the vHIL co-simulation across simulation step sizes (10–1000 μs) and vECU-to-FIL Node composition ratios.

Three dominant trends are observed. First, wall-clock time increases nonlinearly as step size decreases, with the sharpest growth in the 10–20 μs range. This behavior reflects the increasing number of FMU synchronization steps and socket-level exchanges per unit virtual time, and can be approximated by the following approximation:

$$T_{\text{wall}} = a + \frac{q}{x^p}, \quad x = \text{step size} \quad (4)$$

For the fully vECU-only configuration (four Renode vECUs), a 5-second simulation required 620.6 ± 8.1 s of wall-clock time at a 10 μs step size, making sub-20 μs operation impractical for multi-vECU scenarios.

Second, the impact of adding vECUs is disproportionately large at small step sizes. In the 10–20 μs region, each additional vECU introduces cumulative socket-communication and event-scheduling overhead per step. Incorporating a per-step communication term yields the following extended approximation:

$$T_{\text{wall}} = a + t \cdot n \cdot \frac{b}{x} + \frac{q}{x^p}, \quad x = \text{step size}, n = \text{number of vECUs} \quad (5)$$

where t is the fixed socket-communication cost incurred per vECU at each simulation step, n is the number of Renode vECUs in the configuration, $b = 5$ s is the simulation duration (fixed), and b/x is the total number of simulation steps. At step sizes above 200 μs , the socket overhead per step becomes proportionally smaller relative to the total step budget, so increasing the number of vECUs does not significantly increase the total wall-clock time. The dominant component of t is the blocking synchronization handshake between each Renode vECU and MasterSim, which is architecturally required for simulation-time determinism; replacing the socket transport with shared-memory IPC would reduce data-transfer latency but would not eliminate this handshake, leaving the scalability limit largely unchanged.

The parameters of both equations were estimated by nonlinear least-squares regression (`scipy.optimize.curve_fit`) applied to the measurement data described in Table 8. Equation (4) was calibrated on the fully vECU-only configuration ($n = 4$), yielding $a = 10.70$ s, $q = 10,253$, $p = 1.225$ ($R^2 = 0.998$). Equation (5) was fitted jointly across all four node-count configurations (28 measurement points), yielding $a = 8.46$ s, $t = 1.39 \times 10^{-4}$ s step $^{-1}$ vECU $^{-1}$, $q = 8504$, $p = 1.419$ ($R^2 = 0.999$). Here a captures fixed per-session overhead independent of step size (FMU initialization and master scheduling); t quantifies the per-vECU blocking socket-handshake latency per simulation step; and q and p together characterize the superlinear growth in synchronization cost as step size decreases.

Third, and most significantly for scalability, configurations that include FIL Nodes in place of one or more vECUs achieve substantially lower and more stable wall-clock times, as shown in Figure 9. Because FIL Nodes execute as self-contained FMUs without socket-based external synchronization, they do not contribute per-step communication overhead. As a result, replacing vECUs with FIL Nodes for network-communication processing removes the dominant scaling factor, enabling multi-node co-simulations to operate at finer step sizes without prohibitive runtime cost. Notably, this scalability benefit is most pronounced at small step sizes, where socket-synchronization overhead dominates; at larger step sizes, the benefit diminishes as the per-step overhead becomes a smaller fraction of the total simulation cost.

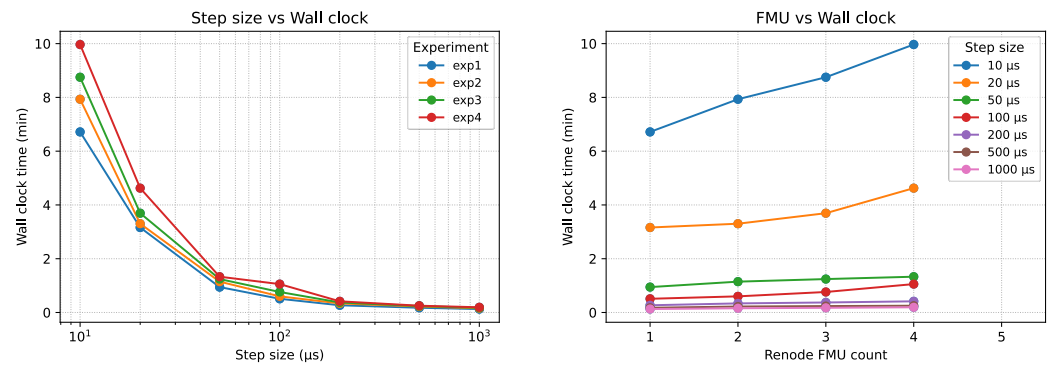


Figure 9. Wall-clock execution time for a 5-second vHIL simulation under different node compositions. Left: wall-clock time vs. step size for varying vECU counts (FIL Node configurations shown for comparison). Right: wall-clock time vs. node count for selected step sizes. FIL Node configurations (dashed) remain significantly below the vECU-only baseline (solid).

To directly validate FIL Node scalability at a 4-node scale, experiment exp5 (0 vECU + 4 FIL Nodes) uses the same total node count as exp4 but replaces all vECUs with FIL Nodes, enabling a direct comparison of the two node types at equal scale. A supplementary 8-FIL-Node measurement (0 vECU + 8 FIL Nodes) additionally doubles the FIL Node count to assess count-induced overhead. Six repeated runs yielded: exp5 = 0.445 ± 0.017 s; the 8-FIL-Node baseline = 0.750 ± 0.011 s.

Two conclusions follow. First, replacing four vECUs with four FIL Nodes (exp4 vs. exp5, equal node count) reduces wall-clock time from 11.60 ± 0.32 s to 0.445 ± 0.017 s—a $26.1\times$ reduction—confirming that the FIL Node architecture eliminates the dominant Renode socket-synchronization cost. Second, doubling the FIL Node count from four to eight (exp5 vs. the 8-FIL-Node baseline) increases wall-clock by only 0.305 s (+69%), from 0.445 s to 0.750 s. The per-node incremental cost is approximately 0.076 s per FIL Node, compared to 1.304 s per vECU—a $17.1\times$ difference. Consequently, in any mixed vHIL configuration that includes at least one vECU, the overhead of additional FIL Nodes is entirely masked by the vECU blocking time. Structurally, this is captured by Equation (5), which contains no FIL-Node-count term; setting $n = 0$ confirms that the FIL Node count does not appear as a scaling factor in the wall-clock model. These results confirm that the framework scales to 8-node configurations representative of production automotive CAN segments (typically 4–12 nodes per segment in practice, as determined by functional partitioning and bus-load constraints; [24] specifies an electrical maximum of 32 nodes per segment based on bus capacitance and termination requirements, with practical deployments falling well within this bound) without throughput degradation.

Because the wall-clock difference between vECU-containing and FIL-only configurations is so pronounced—spanning more than an order of magnitude at coarser step sizes—the two categories are not directly comparable on the same axis. Accordingly, Table 9 reports three configurations separately: exp4 (4 vECU + 0 FIL), exp6 (8 vECU + 0 FIL), and exp5 (0 vECU + 4 FIL), allowing the vECU scaling effect and the FIL Node advantage to be read independently across all seven step sizes. As the step size decreases from 1000 μs to 10 μs, both vECU-containing configurations (exp4 and exp6) require more wall-clock time due to the increase in total simulation steps; however, exp5 remains consistently and substantially faster at every step size. The exp6 column illustrates the vECU scaling behavior: doubling the vECU count from four to eight increases wall-clock time by $1.07\times$ at 1000 μs and up to $2.47\times$ at 50 μs, confirming that the socket-synchronization overhead accumulates with both vECU count and step frequency. The speedup ratio between exp4 and exp5 narrows at finer step sizes ($26.1\times$ at 1 ms, $3.7\times$ at 10 μs) because the per-step computation of the FIL Nodes

themselves becomes non-negligible at 500 000 steps per simulation, whereas in vECU-containing configurations, the Renode session overhead and socket-synchronization cost grow proportionally with step count from the outset. This behavior confirms that FIL Node wall-clock time scales as a function of step count alone, without the fixed Renode-session initialization overhead present in every vECU-containing configuration.

Table 9. Wall-clock time comparison: exp4 (4 vECU + 0 FIL), exp5 (0 vECU + 4 FIL), and exp6 (8 vECU + 0 FIL) across step sizes.

Step Size	exp4: 4 vECU (s)	exp5: 4 FIL (s)	exp6: 8 vECU (s)
10 μ s	620.6	166.3	1389.7
20 μ s	277.5	51.3	561.6
50 μ s	79.8	13.5	197.3
100 μ s	46.2	5.53	91.5
200 μ s	24.9	3.00	47.0
500 μ s	15.1	1.09	20.9
1000 μ s	11.60	0.445	12.4

It is worth noting that the FIL-only configuration exp5 (and the supplementary 8-FIL-Node measurement), while useful as computational benchmarks, do not represent a practically meaningful operational scenario within the scope of this work. The proposed framework is fundamentally oriented toward virtual ECU software validation: FIL Nodes model network communication behavior exclusively and do not execute any ECU application software, so a configuration devoid of vECUs does not fulfill the framework's central verification role. These FIL-only measurements therefore serve a strictly analytical purpose—quantifying the wall-clock cost in the complete absence of Renode–MasterSim socket-synchronization overhead, and thereby demonstrating how substantially that overhead dominates execution time in any configuration containing at least one vECU.

These findings have two practical implications. First, step size is the dominant parameter governing computational cost in vHIL simulation, and single-vECU configurations can be operated at step sizes as fine as 10 μ s without excessive runtime. Second, FIL Nodes act as a structural scalability buffer: by offloading CAN communication into independent FMUs, they extend the practical operating range of the vHIL environment toward finer step sizes and larger node counts. This structural advantage is independent of DBC content and applies to any network configuration generated by the Auto Generator pipeline.

5.8. Engineering Effort Analysis

A stated motivation of the Auto Generator pipeline is to reduce engineering effort by eliminating manual DBC-to-FMU configuration. To provide quantitative evidence for this claim, we use error-prone operations as a proxy metric: points in the workflow where an engineer must transcribe a value from the DBC specification into the FMU configuration by hand, and where a transcription mismatch silently produces an incorrect FMU without a compile-time error.

Three categories of error-prone operation are identified in manual DBC-to-FMU configuration. First, Message ID hardcoding: each CAN message identifier must be entered as a hexadecimal literal, yielding one error-prone entry per message (N entries for N messages). Second, signal scaling parameters: each signal requires an offset value and a scale value to be entered separately ($2K$ entries for K signals in the DBC file). Third, signal bit-field parameters: each signal requires a start-bit position and a bit-length value ($2K$ entries). The total number of error-prone operations in manual configuration is therefore

$$E_{\text{manual}} = N + 4K \quad (6)$$

where N is the number of CAN messages and $K \geq N$ is the total number of signals defined in the DBC file. For the 4-ECU DBC file used in the experiments ($N = 25$ messages, $K = 51$ signals), Equation (6) yields $E_{\text{manual}} = 25 + 4 \times 51 = 229$ error-prone operations. The Auto Generator derives all three categories directly from the DBC schema — message identifiers, scaling parameters, and bit-field definitions are parsed from the structured DBC representation rather than transcribed by hand — reducing the error-prone operation count to zero ($E_{\text{auto}} = 0$) regardless of DBC size.

Equation (6) also provides a scaling argument: for a 16-node network with a typical DBC file of $N = 100$ messages and $K = 400$ signals, manual configuration would entail $100 + 4 \times 400 = 1700$ error-prone operations, all eliminated by the Auto Generator. We note that operation count is a proxy for actual engineering effort and does not capture the speed difference between experienced and novice engineers; it nonetheless provides an objective, DBC-size-dependent lower bound on the manual configuration burden that grows linearly with network complexity.

For comparison, the Vector CANoe Interaction Layer (IL) achieves $E = 0$ error-prone operations and under one minute of configuration time when an OEM-supplied DLL is available, as the DLL encodes DBC-derived transmission semantics automatically and the manual steps are reduced to verification only. Without an OEM DLL, the CANoe IL workflow requires manual transmission-scheduling configuration comparable to the Simulink Vehicle Network Toolbox, carrying the same class of transcription error risk. The FIL Node Auto Generator achieves $E_{\text{auto}} = 0$ without any external DLL dependency, deriving all message IDs, scaling parameters, and bit-field definitions directly from the DBC schema and applying equally to any DBC-defined network.

6. Conclusions

This paper presented a configuration-driven virtual HIL (vHIL) methodology that integrates DBC-based FMU Interaction Layer (FIL) Nodes, an automated generation framework, and Renode-based virtual ECUs to reduce network-configuration effort and early integration delays in conventional HIL workflows. As automotive systems increasingly adopt multi-ECU and network-centric architectures, manual mapping of DBC-defined messages and signals remains a major contributor to test-readiness time and integration errors. The proposed approach directly addresses this bottleneck by externalizing network interaction behavior into automatically generated, FMI-compliant components.

A key contribution of this work is the design of the FIL Node as an explicit interaction layer that decouples CAN communication semantics from ECU execution. By statically analyzing DBC files and test specifications, the proposed methodology constructs a reusable Input–Logic–Output communication skeleton that supports both cyclic and event-triggered traffic without manual configuration. This design is operationalized through an Auto Generator framework that converts DBC files into a message-oriented intermediate representation, derives node-level transmission behavior, generates model skeletons and C/H source code, and exports FIL Node FMUs in a consistent and reproducible manner.

Experimental validation confirmed that the proposed vHIL environment preserves timing fidelity relative to physical ECU measurements. The Renode-based vECU reproduced the physical ECU message transmission period (reference: 100.010 ms) with a maximum relative error of 0.171% across all evaluated step sizes and a minimum of 0.086% at 10 μ s, confirming binary-level timing retention. Run-to-run variation across five repeated measurements was below 0.01 ms for all configurations, demonstrating high reproducibility. Scalability experiments showed that the fully vECU-only configuration required approximately 10 min of wall-clock time to simulate 5 s at a 10 μ s step, while replacing three of four vECUs with FIL Nodes reduced this to approximately 6 min 44 s (a 33% reduction)

and stabilized execution time across node counts—confirming that FIL Nodes function as a structural scalability buffer that removes the dominant socket-synchronization bottleneck. An FIL-only experiment (exp5: 0 vECU + 4 FIL) and a supplementary 8-FIL-Node measurement validated FIL Node scalability: replacing four vECUs with four FIL Nodes reduced wall-clock by $26.1 \times$ ($11.60 \text{ s} \rightarrow 0.445 \text{ s}$), and doubling the FIL Node count from four to eight added only 0.305 s —confirming that FIL Nodes impose $17.1 \times$ lower per-node overhead than vECUs. A further experiment (exp6: 8 vECU + 0 FIL) confirmed that doubling the virtual ECU count to eight yields $\text{ErrorTestFails} = 0$ across all step sizes and approximately linear wall-clock scaling, validating framework scalability to 8-node configurations without throughput degradation. Engineering effort quantification (Section 5.8) showed that manual DBC-to-FMU configuration for the 4-ECU network entails $N + 4K = 229$ error-prone transcription operations; the Auto Generator reduces this count to zero regardless of DBC size, with pipeline execution time of 66.95 s for the same network.

The current study has several limitations. The experimental scenarios were restricted to a single CAN network with up to four ECU nodes and simplified plant models. Realistic vehicle systems involve multiple communication technologies—such as LIN and Automotive Ethernet [25]—as well as gateway routing, diagnostics (UDS), and security mechanisms not yet covered by the Auto Generator pipeline.

Architecturally, extending the framework to additional bus types such as LIN or Automotive Ethernet would require two parallel lines of work: (i) adding a protocol-specific peripheral interface in Renode (e.g., a LIN controller or an Ethernet MAC model) and a corresponding bus FMU in MasterSim, and (ii) adapting the Auto Generator preprocessing stage to parse the relevant network description format (e.g., LDF for LIN, or ARXML for Ethernet). The core Input–Logic–Output structure of the FIL Node and the three-stage build pipeline would remain applicable, as they are not CAN-specific; only the message-format parsing and bus-delay modeling components would need to be replaced. However, developing and validating the required interfaces for each additional bus type is a substantial undertaking in itself—comparable in scope to the CAN-focused work presented in this paper—and is therefore deferred to future work. The evaluation was also limited to single-host execution; extending the framework to distributed co-simulation and CI/CD integration remains an additional direction for future work.

Despite these limitations, the proposed configuration-driven FIL Node methodology demonstrates that network-level automation can be incorporated into vHIL environments without sacrificing timing fidelity or determinism. By eliminating repetitive manual signal mapping and enabling earlier, reproducible multi-ECU integration testing, the approach strengthens shift-left verification and provides a scalable foundation for virtual validation of increasingly complex automotive ECU networks.

Author Contributions: H.L. wrote the entire manuscript and performed model design, software implementation, and formal analysis; H.K. assisted in related research and analysis; J.C. was the principal investigator and is the corresponding author. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Korea Planning & Evaluation Institute of Industrial Technology (KEIT) grant funded by the Ministry of Trade, Industry and Resources (MOTIR) (No. RS-2024-00404601.); Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (MOTIE) (RS-2024-00415938, HRD Program for Industrial Innovation)

Data Availability Statement: The source code of the FIL Node Auto Generator is publicly available at <https://github.com/HarimLee-kr/autofil-dbc-fmu-generator> (accessed on 5 June 2026). The DBC network definition files and raw experimental measurement data used in this study are not publicly available due to proprietary restrictions of the collaborating organization. Data are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ADAS	Advanced Driver-Assistance Systems
ECU	Electronic Control Unit
HPC	High-Performance Computing
HIL	Hardware-in-the-Loop
vHIL	Virtual Hardware-in-the-Loop
vECU	Virtual ECU (Virtual Electronic Control Unit)
UUT	Unit Under Test
CAN	Controller Area Network
AUTOSAR	AUTomotive Open System ARchitecture
FMI	Functional Mock-up Interface
DBC	CAN DataBase (file)
IL	Interaction Layer

References

1. Esfandyari, M. J.; Reza, M.; Esfahanian, V.; Nehzati, H. A Low-Cost Single PC Based HiL Simulation Platform for Verification of the Hybrid Electric Vehicle Control Unit. In Proceedings of the Iranian Conference on Electrical Engineering (ICEE), Mashhad, Iran, 8–10 May 2018; pp. 760–765. <https://doi.org/10.1109/icee.2018.8472636>.
2. Hanselmann, H. Hardware-in-the-Loop Simulation Testing and Its Integration into a CACSD Toolset. In Proceedings of the IEEE Conference on Computer-Aided Control System Design (CACSD), Dearborn, MI, USA, 15–18 September 1996. <https://doi.org/10.1109/cacsd.1996.555253>.
3. Mikelsons, L.; Samlaus, R. Towards Virtual Validation of ECU Software Using FMI. In Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, 15–17 May 2017; Volume 132, pp. 307–311. <https://doi.org/10.3384/ecp17132307>.
4. Kyung, D.-G.; Joe, I.; Kim, W. An FMI-based Time Management Scheme for Real-time Co-Simulation. *J. IEEE* **2020**, *24*, 426–434. <https://doi.org/10.7471/IKEEE.2020.24.2.426>.
5. Kim, H.; Lee, H.; Cho, J. A FMI-Based Approach for CAN Bus Simulation and Simulink Model Integration in Vehicle Simulation Environment. In Proceedings of the 14th IEEE International Conference on ICT Convergence (ICTC), Jeju Island, Republic of Korea, 11–13 October 2023; pp. 279–284. <https://doi.org/10.1109/ICTC58733.2023.10393858>.
6. Kim, H.; Lee, H.; Cho, J. A Coordination Layer for Time Synchronization in Level-4 Multi-vECU Simulation. *Electronics* **2025**, *14*, 1690. <https://doi.org/10.3390/electronics14081690>.
7. Bertsch, C.; Jumayev, K.; Junghanns, A.; R., P.; Menne, B.; Najafi, M.; Pfitzer, T.; Ribbe, J.; Schuch, K.; Süvern, M.; Täuber, P. FMI Layered Standard for Network Communication: Applications in Networked ECU Development. In Proceedings of the 16th International Modelica & FMI Conference, Lucerne, Switzerland, 8–10 September 2025; Volume 218. <https://doi.org/10.3384/ecp218693>.
8. Junior, J. C. V. S.; Brito, A. V.; Nascimento, T. P. Testing Real-Time Embedded Systems with Hardware-in-the-Loop Simulation Using High Level Architecture. In Proceedings of the Brazilian Symposium on Computing Systems Engineering (SBESC), Foz do Iguaçu, Brazil, 3–6 November 2015; pp. 142–147. <https://doi.org/10.1109/sbesc.2015.34>.
9. Mihalič, F.; Truntič, M.; Hren, A. Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges. *Electronics* **2022**, *11*, 2462. <https://doi.org/10.3390/electronics11152462>.
10. Shylla, D.; Jain, A.; Shah, P.; Sekhar, R. Model-in-the-Loop (MIL), Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) Testing in MBD. In Proceedings of the IEEE International Conference on Green Computing and Communication Technologies (GCAT), Bengaluru, India, 6–8 October 2023; pp. 1–6. <https://doi.org/10.1109/gcat59970.2023.10353323>.

11. Schlatzer, D.; Schade, N.; Pannek, J. Enabling Credibility for Virtual Testing of Autonomous Vehicles by Adapting DevOps to Hardware-in-the-Loop Simulations. In Proceedings of the IEEE Intelligent Vehicles Symposium (IV), Cluj-Napoca, Romania, 22–25 June 2025; pp. 1862–1867. <https://doi.org/10.1109/iv64158.2025.11097544>.
12. Kim, H.; Kwak, J.; Cho, J. AUTOSAR-Compatible Level-4 Virtual ECU for the Verification of the Target Binary for Cloud-Native Development. *Electronics* **2024**, *13*, 3704. <https://doi.org/10.3390/electronics13183704>.
13. Zhong, J.; Jin, Y.; Lian, G.; Ding, G.; Zhu, C. Efficient Control System Development Using Real-Time Virtual Hardware-in-the-Loop Simulation. In Proceedings of the International Conference on Control, Automation, Robotics and Vision (ICARCV), Singapore, 7–10 December 2010; pp. 2231–2236. <https://doi.org/10.1109/icarcv.2010.5707315>.
14. Rani, V. S.; Ramesh, A.; Deepthi, K.; Reddy, V. R. Shift-Left Testing in DevOps: A Study of Benefits, Challenges, and Best Practices. In Proceedings of the IEEE International Conference on Advances in Computing, Communication and Robotics (ICACRS), Tamil Nadu, India, 11–13 December 2023. <https://doi.org/10.1109/icacrs58579.2023.10404436>.
15. Jawalkar, S. K. Proactive Software Testing: The Shift-Left Approach to Early Defect Detection and Prevention. *Int. J. Multidiscip. Res.* **2019**, *1*, 36366. <https://doi.org/10.36948/ijfmr.2019.v01i02.36366>.
16. Bergström, D.; Göransson, R. Model- and Hardware-in-the-Loop Testing in a Model-Based Design Workflow. Master's Thesis, Lund University, Lund, Sweden, 2016.
17. Koppak, R. Multi-Domain Fault Simulation Using Virtual Prototypes. Ph.D. Thesis, Universität Tübingen, Tübingen, Germany, 2022.
18. Bunzel, S. AUTOSAR—The Standardized Software Architecture. *Inform. Spektrum* **2010**, *34*, 79–83. <https://doi.org/10.1007/s00287-010-0506-7>.
19. AUTOSAR. Available online: <https://www.autosar.org/> (accessed on 14 September 2025).
20. Blochwitz, T.; Otter, M.; Åkesson, J.; Arnold, M.; Clauß, C.; Elmqvist, H.; Friedrich, M.; Junghanns, A.; Mauss, J.; Neumerkel, D.; et al. Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models. In Proceedings of the 9th International MODELICA Conference, Munich, Germany, 3–5 September 2012. <https://doi.org/10.3384/ecp12076173>.
21. Blochwitz, T.; Otter, M.; Arnold, M. G.; Bausch, C.; Elmqvist, H.; Junghanns, A.; Mauss, J.; Monteiro, M. S.; Neidhold, T.; Neumerkel, D.; Olsson, H.; Peetz, J.-V.; Wolf, S.; Clauß, C. The Functional Mockup Interface for Tool Independent Exchange of Simulation Models. In Proceedings of the 8th International Modelica Conference, Dresden, Germany, 20–22 March 2011. <https://doi.org/10.3384/ecp11063105>.
22. Renode. Available online: <https://www.renode.io/> (accessed on 14 September 2025).
23. ISO/FDIS 11898-1; Road Vehicles—Controller Area Network (CAN)—Part 1: Data Link Layer and Physical Signalling. ISO: Geneva, Switzerland. Available online: <https://www.iso.org/standard/86384.html> (accessed on 30 November 2025).
24. ISO/DIS 11898-2; Road Vehicles—Controller Area Network (CAN)—Part 2: High-Speed Medium Access Unit. ISO: Geneva, Switzerland. Available online: <https://www.iso.org/standard/85120.html> (accessed on 30 November 2025).
25. Lo Bello, L.; Patti, G.; Leonardi, L. A Perspective on Ethernet in Automotive Communications—Current Status and Future Trends. *Appl. Sci.* **2023**, *13*, 1278. <https://doi.org/10.3390/app13031278>.

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.