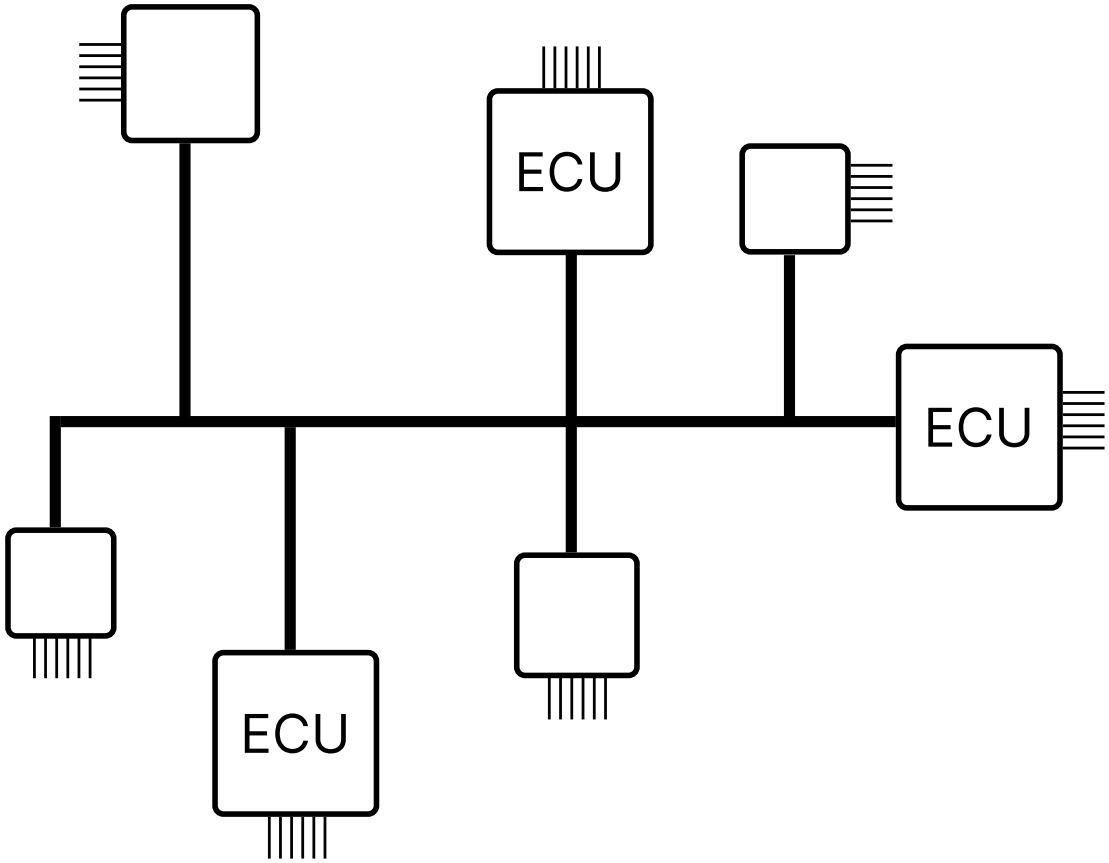
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Parallelisation of Virtual ECUs for Vehicle Simulation

Design and Implementation of FMI 3.0 and FMI-LS-BUS Based CAN Communication for Scalable Automotive System Simulation

Master's thesis in Computer Science and Engineering

Rasmus Johansson
Erik Tran Simonsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2026

Master's thesis 2026

Parallelisation of Virtual ECUs for Vehicle Simulation

Design and Implementation of FMI 3.0 and FMI-LS-BUS Based CAN Communication for Scalable Automotive System Simulation

Rasmus Johansson
Erik Tran Simonsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2026

Parallelisation of Virtual ECUs for Vehicle Simulation
Design and Implementation of FMI 3.0 and FMI-LS-BUS Based CAN Communication for Scalable Automotive System Simulation
Rasmus Johansson
Erik Tran Simonsson

© Rasmus Johansson, Erik Tran Simonsson, 2026

Supervisor: András Kovács, Department of Computer Science and Engineering
Advisor: Niklas Borg, Trucks Technology & Industrial Division
Examiner: Krasimir Angelov, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31-772 10 00

Typeset in Typst
Gothenburg, Sweden, 2026

Parallelisation of Virtual ECUs for Vehicle Simulation
Design and Implementation of FMI 3.0 and FMI-LS-BUS Based CAN Communication for Scalable Automotive System Simulation

Rasmus Johansson
Erik Tran Simonsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Calibration of complex propulsion systems in commercial vehicles increasingly relies on large-scale Co-Simulation of Virtual Electronic Control Units (vECUs). However, existing simulation pipelines are constrained by non-parallel execution and dependencies on host-configured virtual CAN devices, limiting scalability and portability. This thesis presents a scalable, parallelisable Co-Simulation architecture for vECUs using the Functional Mock-up Interface (FMI) 3.0 standard and its FMI-LS-BUS extension for network communication.

We developed a proof-of-concept system that exports vECU models as FMI 3.0 Co-Simulation FMUs with Low-Cut FMI-LS-BUS support, enabling bus-oriented communication without manual signal-level mapping. The implementation bridges Volvo TTI's internal vECU CAN driver with the FMI interface, utilising dynamic generation to derive CAN network terminals from vehicle configuration artefacts. The system leverages the SIL Kit middleware by Vector for inter-process communication, replacing OS-dependent virtual devices with platform-independent messaging. Critical contributions include modifications to the SIL Kit FMU Importer to correctly handle serialised sequences of multiple CAN operations within single FMI binary variables.

Evaluation results demonstrate that the prototype successfully preserves the logical semantics of CAN communication across multiple parallel processes. Benchmarks with up to 100 concurrent FMU participants show linear scalability in per-process runtime when sufficient processor cores are available, with approximately 25% overhead relative to non-communicating simulations in the same benchmark configuration. The automated network mapping eliminates the need for thousands of manual signal connections as participant count grows. While the prototype operates at the logical frame-exchange level without modelling physical bus timing, the results confirm that FMI-LS-BUS is a viable foundation for scalable automotive system simulation.

Keywords: Simulation, Co-Simulation, FMI, Inter-Process Communication (IPC), Controller Area Network (CAN), FMI-LS-BUS, Virtual Electronic Control Unit (vECU)

Acknowledgements

This thesis was carried out in collaboration with Volvo Trucks Technology & Industrial Division. We would like to express our gratitude to our advisor, Niklas Borg, and Christian Axbrink, for their guidance and supervision throughout the work.

We would also like to thank our academic supervisor, András Kovács, for his valuable feedback, support, and insightful discussions during the project, and Krasimir Angelov for acting as examiner.

Rasmus Johansson and Erik Tran Simonsson, Gothenburg, 2026-06-11

Contents

List of Figures

List of Tables

List of Abbreviations

ACK	Acknowledge	**FMI-LS-BUS**	FMI Layered Standard for Network Communication
API	Application Programming Interface	**FMU**	Functional Mock-up Unit
CAN	Controller Area Network	**HIL**	Hardware-in-the-Loop
CAN-FD	Controller Area Network Flexible Data-Rate	**IPC**	Inter-Process Communication
CAN-XL	Controller Area Network Extra Long	**RPC**	Remote Procedure Call
CRC	Cyclic Redundancy Check	**RTR**	Remote Transmission Request
DLC	Data Length Code	**SIL**	Software-in-the-Loop
ECU	Electronic Control Unit	**SOF**	Start of Frame
EOF	End of Frame	**Volvo-TTI**	Volvo Trucks Technology & Industrial Division
FMI	Functional Mock-up Interface	**vECU**	Virtual Electronic Control Unit

1

Introduction

In commercial vehicles, powertrains have grown increasingly complex as manufacturers work to improve energy efficiency and reduce emissions [1]. Modern propulsion systems rely on an increasing number of sensors, actuators, and software-controlled functions, enabling more precise control. However, this also increases the number of parameters to be calibrated, which is costly and time-consuming when done through physical experiments.

At Volvo Trucks Technology & Industrial Division (**Volvo-TTI**), a previous master's thesis demonstrated that methods such as genetic algorithms and Bayesian optimisation could be utilised to address multi-objective calibration problems, where balancing conflicting goals is crucial for identifying suitable parameter settings [2]. However, the existing simulation system, including its current Inter-Process Communication (**IPC**) mechanisms, is limited by its non-parallel execution pipeline.

In addition, the embedded systems sector faces rising demand for virtualised testing and validation. As software becomes more distributed and vehicle functions increasingly rely on interactions between multiple Electronic Control Units (**ECUs**), physical testing alone is no longer sufficient. Virtual simulation platforms enable earlier testing phases, a broader range of scenarios, and reduced dependence on physical rigs or prototype vehicles. This approach is relevant for internal combustion, battery-electric, and fuel-cell electric powertrains, where software-heavy control systems are vital for performance and reliability. Consequently, the ability to run high-throughput simulations is a key enabler of engineering efficiency.

1.1 Problem

Modern commercial vehicles use multiple **ECUs** that communicate over in-vehicle networks such as Controller Area Network (**CAN**). **Volvo-TTI** has an existing simulation environment for Virtual Electronic Control Units (**vECUs**) and their interactions, along with two approaches for inter-model communication.

The first approach relies on manually configured signal mappings between models. Because each signal connection must be specified explicitly, this approach is error-prone and labour-intensive. As the number of **ECUs** and signals increases, it becomes increasingly difficult to maintain and eventually infeasible.

The second approach uses virtual **CAN** buses. This provides a bus-oriented communication abstraction and allows **CAN** messages to be routed without manually specifying each signal connection. However, it requires virtual **CAN** devices to be configured on the host operating system. These devices are global host resources, limited in number, and currently fully supported only on Windows. As a result, this approach restricts portability and limits the number of simulations that can run in parallel.

Together, these limitations make it difficult to scale the existing toolchain and to use compute resources efficiently. A communication approach is therefore needed that preserves **CAN** semantics while avoiding manual signal mapping and host-dependent virtual bus resources.

1.2 Aim

This thesis aims to demonstrate a proof of concept for simulating communicating **vECUs** using the Functional Mock-up Interface (**FMI**) standard with the FMI Layered Standard for Network Communication (**FMI-LS-BUS**) extension for **CAN** communication, which should combine the advantages of the two existing communication approaches without inheriting their drawbacks.

The work designs and implements a scalable architecture that enables **vECUs** to be exported as Functional Mock-up Units (**FMUs**) with support for **FMI** 3.0 and **FMI-LS-BUS**. The exported **FMUs** are intended to be simulated in an external Co-Simulation environment that supports **FMI-LS-BUS** communication.

The thesis investigates how **CAN** messages can be represented and routed between **FMUs** using operating-system-independent communication mechanisms. It also evaluates whether the exported **FMUs** can execute in separate processes while preserving correct communication behaviour.

Correct communication is validated using two **FMUs**, and scalability is evaluated with replicated participants, up to 100 importer processes.

1.3 Limitations

FMI-LS-BUS is a relatively new standard, and there is limited documentation and tooling available. As a result, parts of the implementation require experimentation and iterative development, unlike work based on more established standards.

The proposed architecture is not intended to replace Hardware-in-the-Loop (**HIL**) testing. It supports scalable Co-Simulation of communicating **vECUs**, but it does not provide hard real-time guarantees, since execution depends on host scheduling and computational load. It also does not reproduce hardware-level signal fidelity, such as noise, quantisation, propagation delays, jitter, or electrical effects. Therefore, the approach should be viewed as a software-level Co-Simulation method rather than a substitute for hardware-accurate integration testing.

The prototype operates at the logical **CAN** frame-exchange level. Although **FMI-LS-BUS** supports detailed bus-simulation behaviour, including transmission timing, acknowledgements, bus errors, and arbitration, this requires support from a bus-simulation **FMU** or an importer-integrated bus-simulation. The selected importer does not provide built-in support for such detailed **CAN** bus behaviour, so this thesis evaluates correct frame routing, identifiers, and payload data rather than physical bus timing or fault behaviour.

The work does not include developing a complete **FMI** importer for simulating the **vECUs**. It assumes that an importer with **FMI** 3.0, **FMI-LS-BUS**, and **CAN** bus support is available. However, minor adaptations or implementation-specific constraints of the selected importer may be addressed when they directly affect the prototype. The design and implementation of a complete, general-purpose importer remains out of scope.

2

Background

This chapter briefly introduces the communication standards and the simulation platform utilised in the thesis.

2.1 Controller Area Network (CAN)

Controller Area Network (**CAN**) is a widely used serial communication protocol that enables reliable and efficient data exchange between devices. It operates via a shared bus, allowing multiple devices to send and receive messages without needing a central controller. **CAN** is especially common in the automotive industry, where it connects sensors, actuators, and control modules. **ECUs** connected via **CAN** are physically connected by two wires, namely **CAN** High and **CAN** Low. To keep the bus organised, **CAN** uses a priority-based arbitration system: each message has an identifier, and if two devices try to send at the same time, the message with the highest priority goes through while the others wait. This ensures predictable, collision-free communication even when traffic is heavy, making **CAN** suitable for real-time and safety-critical applications [3].

Figure 1 presents an illustration of a **CAN** bus, in which three **ECUs** handle a frame.

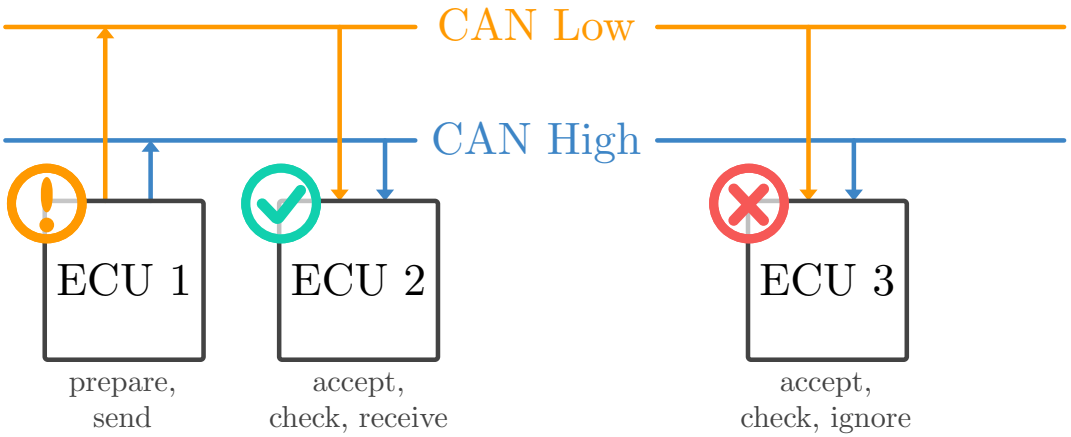


Figure 1 : CAN bus illustration[1]

[1]Reproduced from CSS Electronics, https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial#what-is-can-bus. Used with permission.

2.1.1 CAN-FD and CAN-XL

Newer variants of **CAN** extend the protocol to support higher data throughput and larger payloads [4]. **CAN**, Controller Area Network Flexible Data-Rate (**CAN-FD**), and Controller Area Network Extra Long (**CAN-XL**) share core characteristics and use the same basic bus concept of nodes connected to a central bus. **CAN-FD** extends classic **CAN** mainly by increasing the maximum payload per frame from 8 bytes to 64 bytes, allowing higher bit rates up to 8 Mbit/s, compared to 1 Mbit/s. This makes **CAN-FD** suitable for applications that need more bandwidth while remaining compatible with existing classic **CAN** communication. **CAN-XL** targets even higher performance in vehicle and industrial networks by further increasing payload capacity and data rate. Payloads up to 2048 bytes and data rates up to 20 Mbit/s are supported.

2.1.2 CAN frame layout

A **CAN** message is transferred on the bus using a well-defined frame layout (illustrated in Figure 2), which ensures deterministic arbitration, payload transport, and robust error detection [5]. In classical **CAN**, the two closely related frame types are the data frame, which carries the application payload, and the remote frame, which requests the transmission of a corresponding data frame. Both frame types share the same overall structure and primarily differ in the presence of the data field and the value of the Remote Transmission Request (**RTR**) field.

A **CAN** frame begins with the Start of Frame (**SOF**), which marks the start of transmission and enables bus synchronisation. The following arbitration field contains the message identifier and the **RTR** bit. The identifier defines the message's priority, and arbitration is performed bit-wise, with lower identifier values having higher priority. The control field provides metadata needed to interpret the frame, including the Data Length Code (**DLC**) field, which indicates the payload size and, depending on the identifier format, signalling for standard versus extended identifiers. In a data frame, the data field carries the payload, up to 8 bytes in classical **CAN**, whereas a remote frame omits this field and instead serves as a request for the data associated with the same identifier.

After the payload region, the Cyclic Redundancy Check (**CRC**) field carries the checksum information used by receivers to validate frame integrity. The acknowledgement mechanism is realised in the Acknowledge (**ACK**) field, where receiving nodes confirm a successful **CRC** check. The End of Frame (**EOF**) sequence terminates the frame, and an inter-frame space follows before the next transmission may begin. Together, these fields define a structured message format that supports multi-node communication, prioritised access, and built-in error detection.

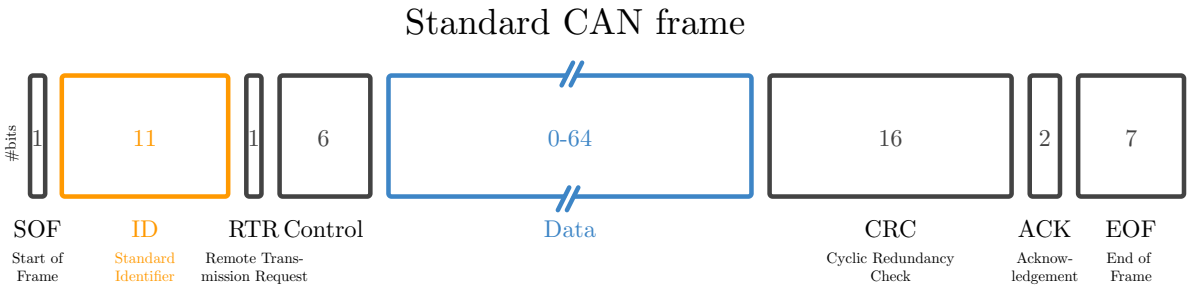


Figure 2 : Standard CAN frame illustration[2]

2.2 Simulation

Simulation is the execution of a model that imitates the behaviour of a real system over time [6]. Rather than interacting directly with the physical system, a simulation generates an artificial time history from a representation and uses that execution to examine system behaviour. In engineering and software-based system development, simulation is especially useful when direct experimentation is expensive, unsafe, impractical, or impossible during early design stages.

A simulation model is not a full reproduction of reality, but an abstraction. It contains only the parts of the system that are relevant to the study objective. This abstraction principle applies in the context of **FMUs** with a defined interface toward an external simulation environment. The **FMU** encapsulates internal behaviour, while selected variables, parameters, and interaction points are exposed to the importer [7].

The state of a system is the set of variables required to describe it at a given point in time relative to the purpose of the study [6]. Simulation then becomes the process of advancing that state over time. State evolution may be continuous, discrete, or hybrid, depending on what is being simulated. Many practical systems combine both types of behaviour, which makes hybrid simulation a natural setting for modern cyber-physical systems.

2.3 Functional Mock-up Interface (FMI)

The **FMI** standard is a free, open, and tool-independent standard for the exchange and simulation of dynamic models between different simulation tools [7]. By encapsulating models in a standardised **FMU**, **FMI** enables component integration while protecting proprietary model details.

An **FMU** is a self-contained unit that encapsulates both the model description and, depending on the interface type, executable simulation logic. It typically includes an XML-based model description, compiled binaries, and optional resources, enabling tool-independent deployment across simulation environments.

[2]Reproduced from CSS Electronics, https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial#can-bus-frames. Used with permission.

The importing environment, often referred to as the importer, is responsible for coordinating one or more **FMUs**. This includes managing time progression, synchronising data exchange between components, and ensuring numerical stability of the overall simulation.

FMI defines multiple interface types to support different integration scenarios:

- In Model Exchange, the **FMU** provides the model equations while the importer handles numerical integration.
- In Co-Simulation, the **FMU** contains its own solver and advances its internal state independently when triggered by the importer.
- Scheduled Execution extends this concept by allowing explicit control over execution order and timing, which is particularly relevant for real-time and embedded applications.

For this work, Co-Simulation is utilised. Figure 3 illustrates a schematic overview of data flow between the user, the Co-Simulation algorithm of the importer, and the FMU for Co-Simulation.

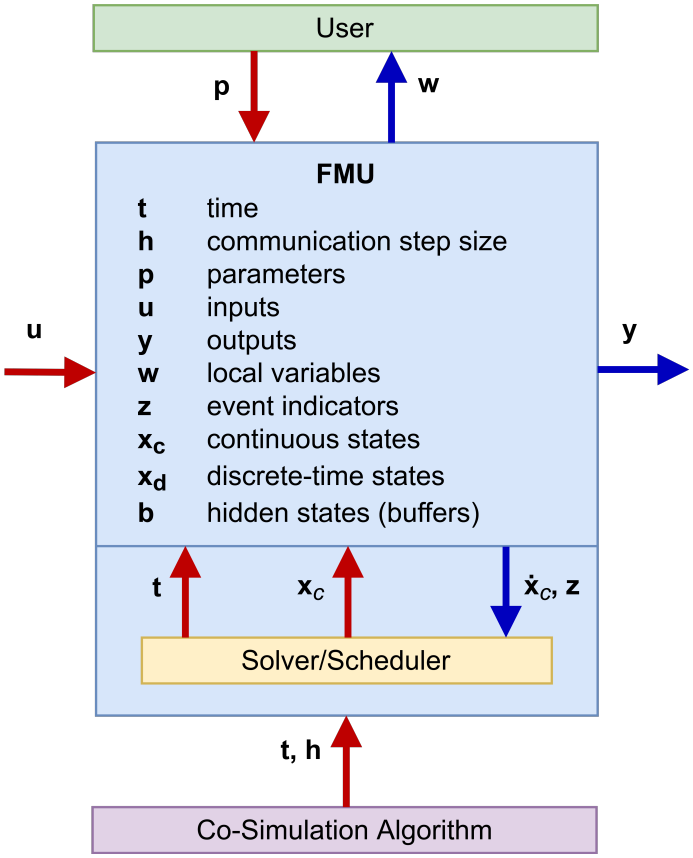


Figure 3 : Co-Simulation data flow illustration[3]

[3]Reproduced from Functional Mock-up Interface Specification, version daae651 (2026-05-05), https://fmi-standard.org/docs/main/#fmi-for-co-simulation-cs © The Modelica Association Project FMI. Licensed under CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/).

2.3.1 FMI-LS-BUS

To extend interoperability to communication aspects, the FMI Layered Standard for Network Communication (**FMI-LS-BUS**) introduces bus-oriented abstractions for data exchange between simulation units [8]. It enables the virtualisation of communication networks used in automotive and beyond, including **CAN**, LIN, FlexRay, and Ethernet.

The standard defines two complementary abstraction levels.

- The Physical Signal Abstraction (High-Cut) focuses on the exchange of signal values with largely idealised network properties (e.g., negligible delay and unlimited bandwidth).
- The Network Abstraction (Low-Cut) emulates a transport layer for protocol-independent communication, allowing more detailed modelling of network behaviour such as bandwidth limitations, arbitration, delays, and message-based data exchange. In Low-Cut **CAN**, communication is represented through clocked binary variables grouped by bus terminals [7]. A transmitting **FMU** exposes a transmission clock and a binary variable containing one or more serialised bus operations. Correspondingly, received traffic is provided through a receive clock and a binary input variable. The exchanged data is therefore not individual application signals, but protocol-level operations such as **CAN** transmission and reception. This makes the abstraction suitable for **vECUs** whose internal behaviour already depends on **CAN** transmission and reception.

The necessary mapping between the standardised **CAN** frame layout used by the existing **vECUs** and the **FMI-LS-BUS** frame layout is illustrated in Figure 4.

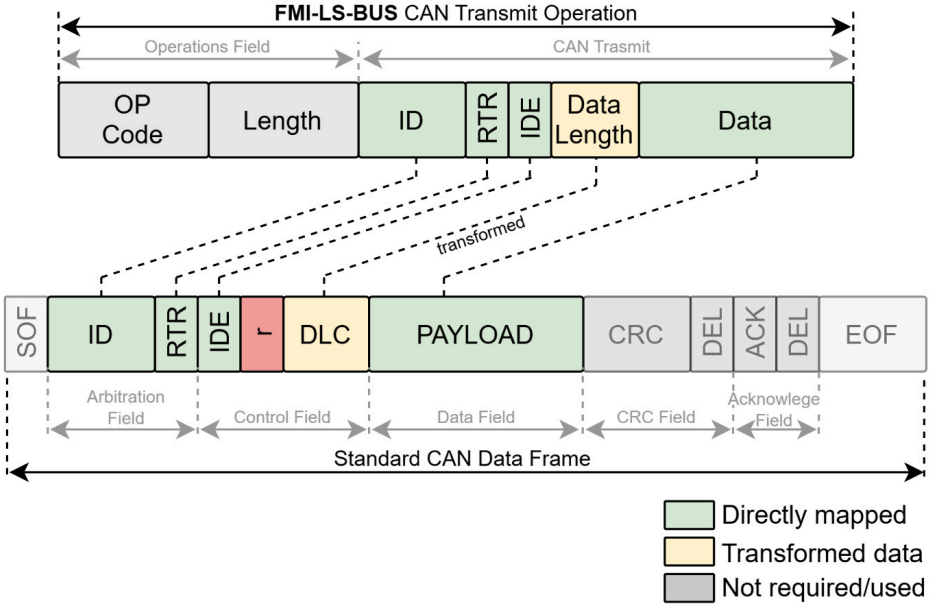


Figure 4 : Mapping between an FMI-LS-BUS CAN transmission bus operation and a standard CAN data frame.[4]

[4]Reproduced from FMI Layered Standard for Network Communication (FMI-LS-BUS), version 2e8c366 (2026-02-13), https://modelica.github.io/fmi-ls-bus/main/#low-cut-layered-standard-

The present work initially targets one Low-Cut **CAN** network and models each exported **FMU** as a network participant rather than as a bus simulation **FMU**. This matches the intended use case: exporting **Volvo-TTI vECUs** that can communicate over a virtual **CAN** bus in an external Co-Simulation environment.

A networked **FMU** using **FMI-LS-BUS** shall support event mode. The communication points on the bus are defined by the clocks associated with the transmitted and received binary variables. For Co-Simulation, this allows the importer or bus simulation to coordinate when network data is exchanged relative to the execution of each **FMU**. In practice, the standard provides the interface structure, while the actual bus behaviour is realised by the surrounding simulation setup, for example, an importer with integrated bus simulation.

2.3.2 Event mode and clocked communication

Event mode is important in **FMI** 3.0 network communication. In Co-Simulation, the **FMU** executes its internal model during `fmi3DoStep`, while discrete communication-related updates are handled when the importer enters event mode. For **FMI-LS-BUS CAN**, this is the natural place to exchange binary bus operations and update communication clocks.

For outgoing traffic, the **FMU** may detect during a communication step that one or more **CAN** frames should be transmitted. Rather than exposing such changes immediately, the **FMU** buffers the corresponding bus operations and signals this through its transmission clock at the end of the step. The importer can then read the associated binary data and route it to other participants. Incoming traffic follows the opposite direction: the importer activates the receive clock, provides the received binary data, and the **FMU** processes it during event handling. This separation fits importer-driven Co-Simulation and avoids mixing protocol handling with ordinary numerical stepping.

FMI-LS-BUS supports both time-based and triggered transmission clocks. Clocks synchronise discrete events between the importer and **FMUs**. With time-based transmission clocks configured to count down, an **FMU** announces a future transmit time before the step starts. The importer adapts the communication step size so all networked **FMUs** hit that time, enter event mode, and exchange the bus operation. With triggered transmission clocks, the **FMU** instead signals pending transmit data only when returning from `fmi3DoStep`, which means bus communication points cannot be scheduled independently and therefore coincide with the importer's regular communication points. In this thesis, triggered clocks are used. This was primarily a practical design choice: the selected importer, **SIL** Kit, supports triggered clocks for **FMI-LS-BUS CAN**, and it provides a direct mapping from when a frame becomes available to exporting it at the end of the cycle.

bus-protocol © The Modelica Association Project FMI. Licensed under CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/).

2.4 SIL Kit

SIL Kit by Vector is an open-source, platform-independent middleware for connecting Software-in-the-Loop (**SIL**) applications and enabling communication between simulation participants [9]. It provides services for common in-vehicle communication technologies, including **CAN**, **CAN-FD**, **CAN-XL**, Ethernet, FlexRay, and LIN, as well as generic **IPC** mechanisms such as publish/subscribe messaging and Remote Procedure Calls (**RPCs**).

A **SIL** Kit-based simulation is typically composed of a registry process and one or more participants. The **SIL** Kit Registry must be started first and acts as a connection broker: participants connect to the registry to discover each other and establish the communication channels required for the simulation. The participants implement the actual simulation logic and interact via **SIL** Kit services (e.g., vehicle network controllers, publish/subscribe topics, and **RPC** endpoints). For coordinating multi-component setups, **SIL** Kit also supports orchestration mechanisms such as synchronised start-up behaviour and a shared virtual simulation time among participants. A **SIL** Kit simulation supports multiple simultaneous communication buses and networks within a single setup.

Participants are created and initially configured by the developer via the **SIL** Kit Application Programming Interface (**API**), while the user can adapt runtime behaviour via YAML-based configuration files provided at start-up. The registry can likewise be configured via a configuration file, enabling flexible reconfiguration of the simulation setup without changing participant source code.

2.4.1 SIL Kit FMU Importer

The **SIL** Kit **FMU** Importer is an extension that allows **FMUs** to be imported and instantiated as **SIL** Kit participants [10]. The importer is executed headlessly and is configured via configuration files passed at launch. In this thesis, **FMUs** are integrated into a distributed Co-Simulation setup using **SIL** Kit as the communication backbone. At the time of writing, the **SIL** Kit **FMU** Importer appears to be the only open-source importer listed on the official **FMI** tools page that supports **FMI-LS-BUS** (based on filtering the tools list by FMI-LS-BUS and open-source license) [11].

3

Method and Implementation

This chapter describes the methodology and implementation. First, the workflow and process are outlined. Second, the overall architecture and design choices are presented at a higher level. Finally, the specifics of the implementation are described.

3.1 Workflow

The work was carried out as an iterative design-and-implementation study. It began with a review of the **FMI** 3.0 and **FMI-LS-BUS** specifications, followed by practical experimentation with existing **FMUs**, Python-based **FMI** tooling, FMPy, and the **SIL** Kit **FMU** Importer. The purpose of this initial phase was to establish a working understanding of the standard, identify which parts were required for a Low-Cut **CAN** proof of concept, and determine which importer could realistically be used during development.

Based on this exploration, the implementation problem was narrowed to a concrete and testable scope: export two **Volvo-TTI vECUs** as Co-Simulation **FMUs** with **FMI** and **FMI-LS-BUS CAN** support, and validate using an importer with built-in **FMI-LS-BUS CAN** support. The importer selection was treated as a method decision rather than as an implementation detail, since it constrained which communication mechanisms could be exercised in practice. **SIL** Kit was selected because it is open source, supports **FMI** 3.0, **FMI-LS-BUS**, and **CAN**. Moreover, it allows isolated proof-of-concept experiments without requiring integration with the full existing simulation platform.

Development then proceeded in small increments. This reduced integration risk and enabled validation of each layer separately: manifest generation, terminal description, model description generation, and **FMI** C **API** functionality.

Once a working build capable of generating **FMUs** with functional Co-Simulation communication was achieved, further improvements were considered, including optimisation and the dynamic handling of multiple **CAN** networks.

3.2 Simulation architecture

The simulation architecture integrates the **FMI** 3.0 interface directly into **Volvo-TTI**'s existing **vECU CAN** communication stack to avoid redundant functionality. The implementation reuses their **CAN** driver's frame buffer, allowing both native and Co-Simulated execution to follow the same data paths.

At each simulation step, the **vECU** produces outgoing **CAN** frames and places them in a transmission queue. These frames are then exposed through the **FMI** 3.0 interface at synchronisation points. A dedicated module, `canBridge.c`, connects the **CAN** driver and the **FMI** 3.0 layer by retrieving outgoing frames, forwarding incoming frames to the **CAN** driver, and notifying the importer when new data is available.

At the Co-Simulation level, the **SIL** Kit **FMU** Importer handles communication between **FMUs** by transferring **FMI** 3.0 variable representations of frames.

Figure 5 presents an overview of the Co-Simulation architecture, in which two **FMUs** handle frames from two networks, A and B.

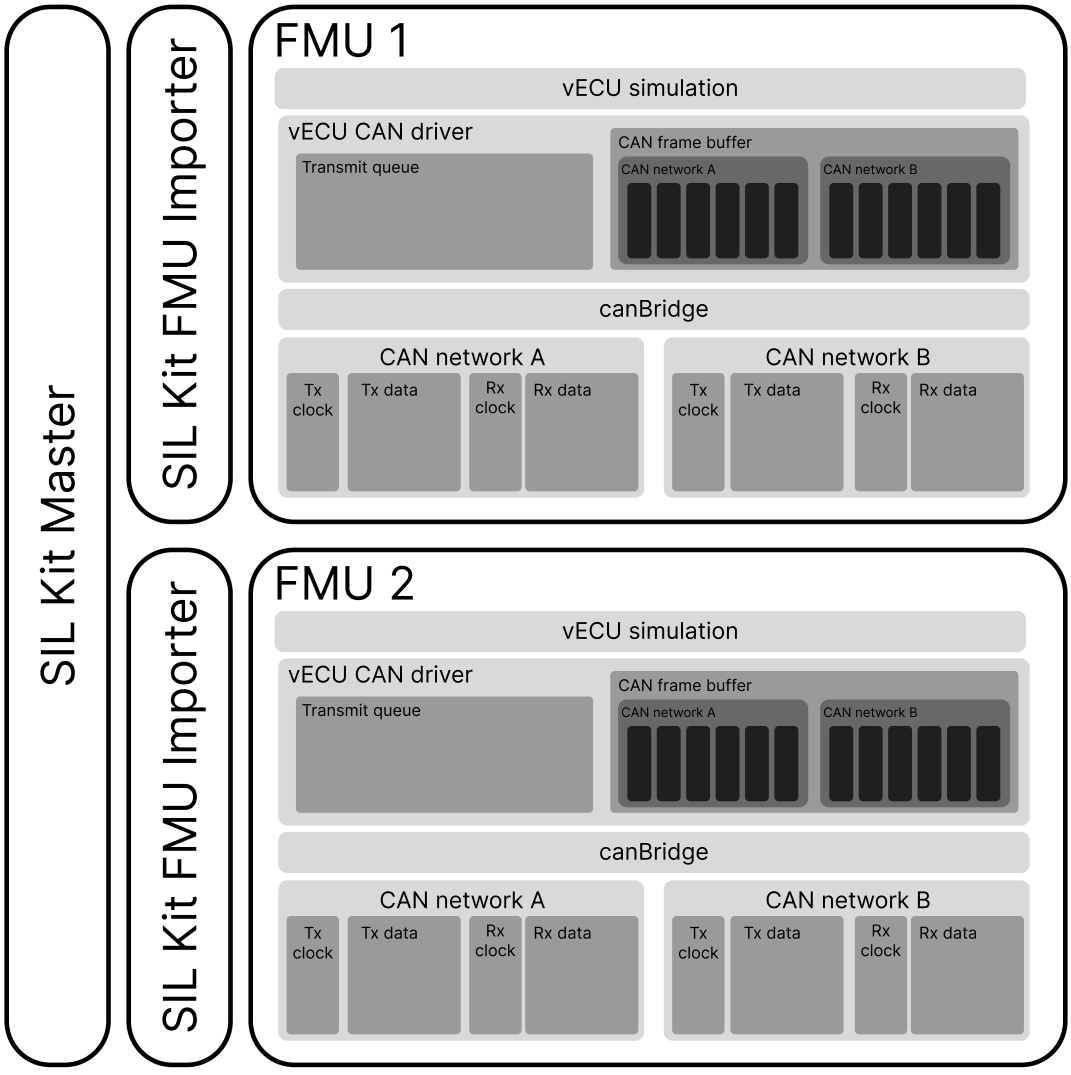


Figure 5 : Overview of the simulation architecture

3.3 Implementation

A small set of explicit design choices guided the implementation.

- First, the **FMU** was treated as a networked **ECU** rather than as a simulation of the **CAN** bus itself.
- Second, only Low-Cut **CAN** was implemented, since the goal was to support virtual **CAN** driver behaviour rather than signal-level exchange.
- Third, triggered transmission clocks were selected over countdown-based clocks, primarily since they matched the capabilities of the chosen importer and simplified early testing.

The implementation itself was divided into two parts: **FMU** interface generation and **FMU** runtime support.

3.4 FMU interface generation

The export pipeline was extended to generate the layered standard manifest, the terminal description file, and the **FMI-LS-BUS**-specific additions to `modelDescription.xml`. In addition to the required **FMI** metadata, the pipeline generates new value references and variable mappings for the **CAN** communication variables, including the transmit and receive clocks and their associated binary variables.

3.4.1 Layered Standard Manifest generation

The export pipeline generates a layered manifest file, `fmi-ls-manifest.xml`, for each exported **FMU** and inserts it into the **FMI-LS-BUS** required archive location. Since all generated **FMUs** represent networked **ECUs** rather than bus simulation components, the same manifest structure is reused for all exports, as shown in Listing 1.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fmiLayeredStandardManifest
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../schema/
    fmi3LayeredStandardBusManifest.xsd"
    xmlns:fmi-ls="http://fmi-standard.org/fmi-ls-manifest"
    fmi-ls:fmi-ls-name="org.fmi-standard.fmi-ls-bus"
    fmi-ls:fmi-ls-version="1.3.0-alpha.1"
    fmi-ls:fmi-ls-description="Layered Standard for the simulation of bus
    communication on a Physical Signal Abstraction or Network Abstraction
    based level."
    isBusSimulationFMU="false"/>
```

Listing 1 : Generated `fmi-ls-manifest.xml`

3.4.2 Terminal description generation

The communication structure is described in `terminalsAndIcons.xml`, which defines one bus terminal for each **CAN** network. Each terminal is named after the network and contains four `TerminalMemberVariable` entries for `Tx_Data`, `Tx_Clock`, `Rx_Data`, and `Rx_Clock`, which reference the corresponding variables in `modelDescription.xml`. An example is illustrated in Listing 2.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<fmiTerminalsAndIcons fmiVersion="3.0">
    <Terminals>
        <Terminal name="NetworkA" description="LS-BUS CAN (Low-Cut) network terminal for NetworkA" terminalKind="org.fmi-ls-bus.network-terminal" matchingRule="org.fmi-ls-bus.transceiver">
            <TerminalMemberVariable variableKind="signal" variableName="NetworkA.Rx_Clock" memberName="Rx_Clock"/>
            <TerminalMemberVariable variableKind="signal" variableName="NetworkA.Rx_Data" memberName="Rx_Data"/>
            <TerminalMemberVariable variableKind="signal" variableName="NetworkA.Tx_Clock" memberName="Tx_Clock"/>
            <TerminalMemberVariable variableKind="signal" variableName="NetworkA.Tx_Data" memberName="Tx_Data"/>
        </Terminal>
        <Terminal name="NetworkB" description="LS-BUS CAN (Low-Cut) network terminal for NetworkB" terminalKind="org.fmi-ls-bus.network-terminal" matchingRule="org.fmi-ls-bus.transceiver">
            <TerminalMemberVariable variableKind="signal" variableName="NetworkB.Rx_Clock" memberName="Rx_Clock"/>
            <TerminalMemberVariable variableKind="signal" variableName="NetworkB.Rx_Data" memberName="Rx_Data"/>
            <TerminalMemberVariable variableKind="signal" variableName="NetworkB.Tx_Clock" memberName="Tx_Clock"/>
            <TerminalMemberVariable variableKind="signal" variableName="NetworkB.Tx_Data" memberName="Tx_Data"/>
        </Terminal>
    </Terminals>
</fmiTerminalsAndIcons>
```

Listing 2 : Generated `terminalsAndIcons.xml` example

3.4.3 `modelDescription.xml` extension

To support the bus terminal definitions, `modelDescription.xml` is extended with the communication variables required by **FMI-LS-BUS**. For each network, the export pipeline generates the corresponding clock and binary variables, assigns value references, and links each binary variable to its clock through the clocks attribute. Clocks and binary variables are coupled through a value-reference linkage. For each bus terminal, the clock variables such as `Tx_Clock` and `Rx_Clock` act as boolean trigger signals, while

the corresponding binary variables such as `Tx_Data` and `Rx_Data` carry the payload. The linkage is established by assigning the clock's value reference to the binary variable's `clocks` field, in accordance with **FMI** 3.0. In this way, an active clock indicates that the associated binary buffer is valid for transmission or reception. At the system level, the variables are linked by name using the terminal identifiers and **CAN** network names, which allows the exported **FMUs** to align with the surrounding simulation environment. This enables coupling with Volvo's existing system without requiring internal modifications.

3.5 Bridging the vECU model with the FMI-LS-BUS interface

The **CAN** bridge connects the internal **vECU CAN** representation to the **FMI-LS-BUS** interface exposed by the **FMU**. Internally, the **vECU** uses its own **CAN** frame representation and driver abstraction. At the **FMI** boundary, however, **CAN** traffic is represented as binary **FMI-LS-BUS CAN** operations, where data buffers contain serialised bus operations and clock variables indicate when the corresponding data is valid. The bridge has two main responsibilities: collecting outgoing **CAN** frames from the **vECU** and encoding them as **FMI-LS-BUS** operations, and receiving **FMI-LS-BUS** operations from the importer and injecting the corresponding **CAN** frames back into the **vECU** model.

3.5.1 Functionality of the CAN interface

On the transmitting side, the bridge collects pending outgoing **CAN** frames via the **vECU CAN** driver interface. Each returned frame is checked before being written to the **FMI-LS-BUS** transmit buffer. The bridge first verifies that the frame contains a valid payload and that the current implementation supports the payload length. While the **FMI-LS-BUS CAN** representation may support larger payload formats, including **CAN** FD and **CAN** XL, this implementation only supports the standard **CAN** payload length. Therefore, if a frame contains more than eight payload bytes, only the first eight are exported, and a warning is logged.

Before writing an outgoing operation, the bridge also checks that there is sufficient space in the target binary buffer. A **CAN** operation is written only if the fully encoded operation fits. This avoids partially written bus operations, which would otherwise prevent the receiving side from reliably parsing the binary data.

The bridge must also ensure that each outgoing frame is placed in the correct **FMI-LS-BUS** buffer. Internally, the **vECU CAN** configuration identifies **CAN** networks using generated network identifiers. The **FMI** interface, however, uses logical network names as the externally visible connection key. The bridge maps the internal network identifier to the corresponding network name, then uses that name to select the correct **FMI-LS-BUS** buffer.

After the target network buffer has been selected, the frame is converted from the internal **vECU CAN** representation into an **FMI-LS-BUS CAN** transmit operation as illustrated in Figure 4. This conversion constructs the required operation header, stores the **CAN** identifier and metadata, copies the payload bytes, and advances the write position in the binary buffer.

On the receiving side, processing is tied to **FMI** event handling. The bridge does not continuously poll the receive buffers. Instead, each channel is checked during the discrete-state update phase, and incoming binary data is processed only if the corresponding receive clock is active.

For each active receive channel, the bridge iterates over the binary buffer, reading one **FMI-LS-BUS** operation at a time. The operation-reading mechanism supports operations of varying size by first checking that an operation header is available, then reading the declared operation length, and finally verifying that the complete operation is present before advancing the read position.

Only **CAN** transmit operations are handled by the current receive implementation. For each operation, the bridge validates that the operation code corresponds to a **CAN** transmit operation and that the encoded payload length is consistent with the available operation size. The operation is then converted from the **FMI-LS-BUS CAN** representation back into the internal **vECU CAN** frame representation as illustrated in Figure 4.

The receive path also performs the reverse network mapping. Incoming data is associated with an **FMI-LS-BUS** channel name, which corresponds to the logical **CAN** network name. The receiving **vECU** may use a different internal network identifier for the same logical network, because network identifiers are generated locally from that **vECU**'s **CAN** artefacts. Therefore, the bridge maps the channel name back to the local network identifier before placing the received frame into the **vECU**'s internal **CAN** memory.

A received **CAN** frame is only copied into the **vECU** if the receiving model contains a corresponding frame buffer. This is not guaranteed. A **CAN** bus may carry frames from many ECUs, whereas an individual **vECU** model typically contains only the frames relevant to that **ECU**'s software. Therefore, an **FMU** may receive a valid **CAN** frame on a connected network without a corresponding local frame representation. Such frames are dropped rather than treated as fatal errors, as is presented in Figure 6.

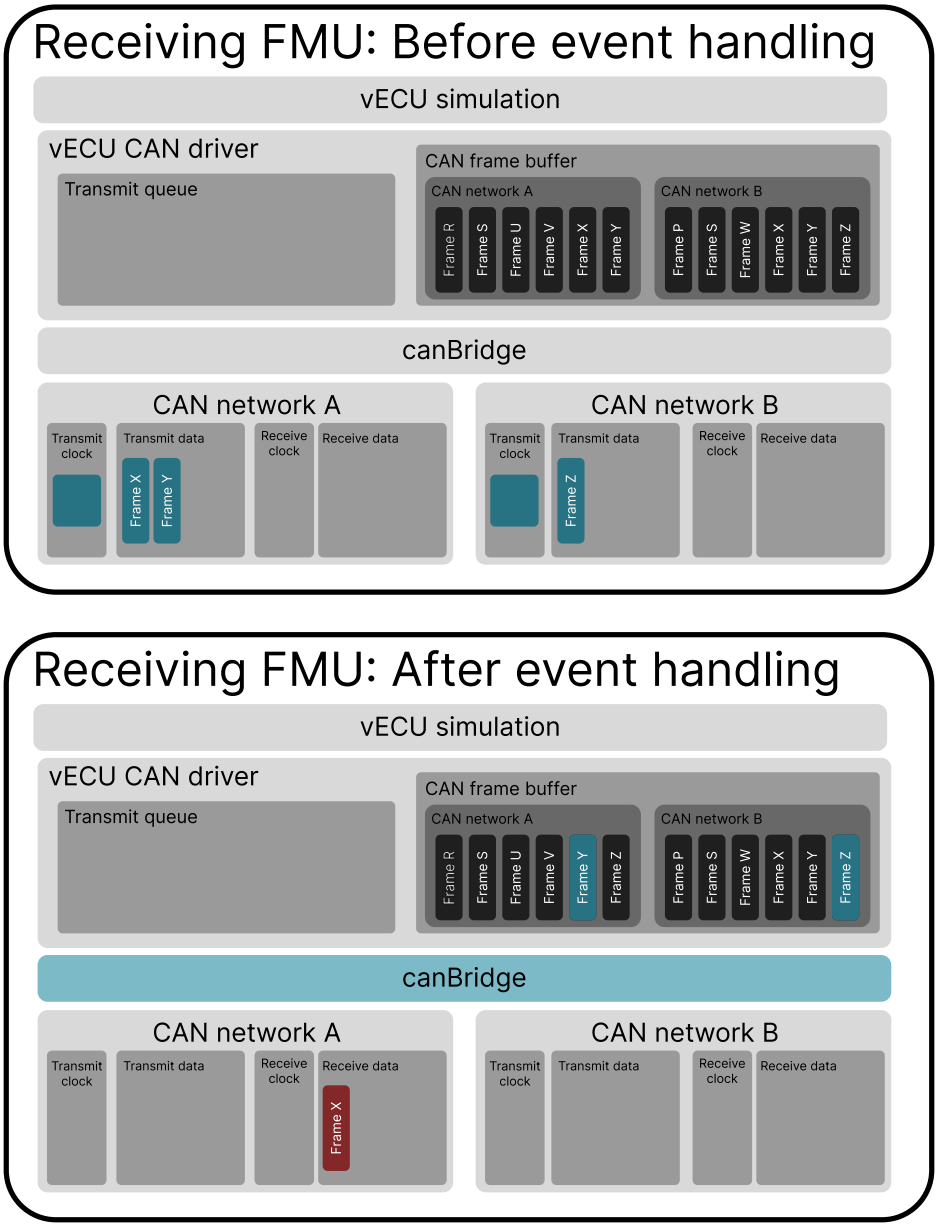


Figure 6 : A CAN frame without a frame representation is dropped

When a matching local frame is found, the bridge copies the received payload into the internal frame memory and notifies the active **CAN** backend through the existing receive path. This allows received **FMI-LS-BUS** traffic to enter the **vECU** through the same conceptual path as **CAN** traffic from other supported backends.

3.5.2 Design choices of the CAN interface

A central design choice in the transmit path is that the bridge drains the pending transmit queue maintained by the **vECU CAN** driver. Each queue entry refers to an internal frame representation, which the bridge uses to retrieve the frame metadata, select the corresponding **FMI-LS-BUS** buffer, and serialise the frame as a **CAN** *Transmit* operation.

The implementation favours the latest available state when several updates occur before the bridge exports the frame. Since the queue references the internal frame representation, the value serialised during event handling is the current frame state at the time of export. This avoids storing multiple copies of the payload in the transmit queue and keeps memory usage bounded. However, intermediate states may be lost if the **vECU** updates the same frame several times before the next communication point. This can be reduced by using a smaller simulation step size, which allows the **FMU** to transmit frames more often, at the cost of additional synchronisation points.

The receive path is designed around **FMI** event semantics. Incoming binary data is processed only when the corresponding receive clock is active. This clock-gated design prevents the bridge from interpreting stale buffer contents and aligns the implementation with the **FMI-LS-BUS** model, where clocks define when data variables are meaningful.

The receive implementation also deliberately parses the message first before notifying the **vECU CAN** backend. This separates external data validation from internal state propagation. The bridge first verifies that the operation is well-formed, that it is an operation type supported by the implementation, that the payload length is valid, and that a local frame representation exists. Only after these checks is the payload copied into the internal frame and the **vECU CAN** stack notified.

For each **CAN** channel, the internal frame representations are stored in order by **CAN** identifier. Therefore, the receive path performs frame lookup using binary search, yielding a lookup complexity of $O(\log n)$ for n represented frames.

Finally, after a received frame has been copied into internal memory, the bridge updates its transmit snapshot for that frame. This prevents a just-received frame from being immediately detected as a new local change and re-exported on the transmitting side. Without this step, an **FMU** could echo received traffic back onto the bus, potentially creating duplicate traffic or feedback loops.

3.6 Extending the vECU driver

To support **FMI-LS-BUS**-based Co-Simulation, the **vECU CAN** driver was extended with an **FMU**-oriented backend. The extension does not replace the existing **CAN** stack, but provides a transport path for internal **CAN** traffic to be staged, collected, serialised, and exchanged via the **FMI** interface. Its role is therefore to connect the

ordinary **vECU CAN** behaviour to the event-based communication model required by **FMI** 3.0.

3.6.1 Functionality of the vECU driver

The extended driver decouples the **vECU** simulation from the **FMI-LS-BUS** transmission. When the **vECU** software transmits a **CAN** frame, the frame is written to the internal **CAN** representation and marked as pending for export. It is not sent through the **FMI** interface immediately. Instead, the **FMU** first completes its current simulation step, after which pending **CAN** traffic is collected and exposed during **FMI** event handling.

The extension provides three main capabilities. First, it allows the bridge to enumerate the **CAN** frames that the **vECU** represents during initialisation. Second, it stages outgoing frames in a pending transmit queue. Third, it supports the receive direction by copying incoming data into the corresponding internal frame and forwarding it through the existing **vECU CAN** receive path.

The pending transmit queue stores references to existing internal frame representations rather than copies of complete **CAN** payloads. When the **vECU** software transmits a frame, the corresponding frame object is marked as pending, and a reference to it is inserted into a fixed-size ring buffer. The internal frame table, therefore, remains the single source of truth, while the queue records which frames require export at the next **FMI** event-handling phase. This avoids scanning all represented **CAN** frames during each collection step. The collection cost instead depends on the number of frames transmitted since the previous collection, which is important when many frames are represented but only a small subset is transmitted during a simulation step.

On the receiving side, incoming frames are matched against the local frame representation. The current implementation uses the ordered structure of the generated frame data to perform efficient lookups via a binary search. If a matching local frame exists, the payload is copied into it and the ordinary receive path is notified. If no matching frame exists, the frame is ignored because a **vECU** is not expected to consume every frame on a shared **CAN** bus.

3.6.2 Design choices of the vECU driver

A central design choice is the two-phase transmit model. The **vECU** may produce **CAN** traffic during its simulation step, but the actual **FMI-LS-BUS** export is deferred until event handling. This follows the **FMI** 3.0 Co-Simulation model and keeps internal model execution separate from importer-controlled communication.

The queue and the **FMI-LS-BUS** buffers are fixed in size. This gives bounded memory usage and avoids dynamic allocation during simulation. It also makes overflow explicit: if the queue or output buffer cannot accept more data, it is reported rather than creating a partially written or inconsistent communication state.

The receive path reuses the existing **vECU CAN** receive mechanism. After a received payload has been placed in the internal frame representation, it is passed upward through the same conceptual path as other **CAN** backends. This avoids introducing **FMI** -specific logic into the upper **CAN** software.

3.7 Dynamic CAN network extraction and interface generation

To automate **CAN** network configuration, the **FMU** interface generation derives the available **CAN** networks directly from the generated **vECU CAN** artefacts. The generation step extracts the set of **CAN** network identifiers from this representation and uses them to construct the **FMI** communication interface.

For each detected **CAN** network, the export pipeline generates one logical **FMI-LS-BUS** terminal. The terminal name is set to the corresponding **CAN** network name, which makes the network name the externally visible connection key. Each terminal is generated with the clocked transmit and receive variables described in Section 3.4.3. The purpose of this section is therefore not to define the clock–data relationship again, but to describe how the required terminal structure is generated from the **vECU CAN** configuration.

At the system level, **CAN** networks are identified by their names. Two **FMUs** exposing a terminal with the same **CAN** network name can therefore be connected to the same logical network by the importer. Data written to the transmit buffer of one participant is routed to the receive buffers of the other participants connected to that network. The routing is based on terminal metadata and **CAN** frame identifiers rather than explicit signal-level connections.

The same network abstraction is reflected in the generated C runtime support. Since the number and names of **CAN** networks are configuration-dependent, the required C structures are generated using templates. The generation step uses Jinja to expand the known set of **CAN** networks into statically allocated C artefacts. This provides configuration-dependent runtime structures without relying on dynamic memory allocation or runtime-sized arrays in C.

The generated C file contains an array of **CAN** network descriptors. Each descriptor represents one **CAN** network and is identified by its network name. The descriptor links this name to the corresponding runtime state and stores references to the associated data buffers, clock variables, and buffer metadata used by the **FMI-LS-BUS** interface. In this way, the generated code provides a static lookup structure that connects the **FMI** -level network terminal to the internal **vECU CAN** representation.

The generated C support also includes functions for iterating over the available **CAN** networks. This allows the runtime bridge to process all generated networks uniformly, independent of how many networks were present in the source configuration. The bridge can therefore traverse the generated descriptors, inspect the associated clocks and

buffers, and route bus operations between the **vECU CAN** runtime and the **FMI-LS-BUS** variables.

This approach makes the mapping between **FMUs** implicit and configuration-driven. Any **vECU FMU** generated from the same **CAN** configuration exposes terminals with identical network names and member structure. Consequently, when multiple such **FMUs** are co-simulated, **CAN** frames can be routed consistently between participants without requiring manual configuration of connections at the frame or signal level.

The method also ensures consistency across all generated layers. Since the **CAN** network names are extracted from the generated **vECU** artefacts, the same identifiers are used in the runtime implementation, the **FMI** model description, the generated C support, and the **FMI-LS-BUS** metadata. Changes to the **CAN** configuration, such as adding, removing, or renaming networks, are therefore automatically propagated to the **FMU** interface and runtime support, without requiring manual configuration duplication.

3.8 Writing binary data to frame buffers

The **FMI-LS-BUS** reference implementation provides the `FMI3_LS_BUS_BUFFER_WRITE` macro for writing data into an **FMU** receive buffer. However, when additional data needed to be placed into an already populated buffer, this macro overwrote the existing contents rather than appending to them. Since the **FMI-LS-BUS** documentation specifies that multiple **CAN** frames may be placed consecutively in a single binary blob [12], an append capability was required. A dedicated macro, `FMI3_LS_BUS_BUFFER_APPEND`, seen in Listing 3, was implemented to copy new data to the current write position, advance the write pointer, and set a status flag, while also performing a bounds check to prevent buffer overflow.

```c
#define FMI3_LS_BUS_BUFFER_APPEND(BufferInfo, Data, DataLength)                \
    do {                                                                       \
        size_t used = (size_t)((BufferInfo)->writePos - (BufferInfo)->start); \
        if (used + (DataLength) <= (BufferInfo)->size) {                       \
            memcpy((BufferInfo)->writePos, (Data), (DataLength));              \
            (BufferInfo)->writePos += (DataLength);                            \
            (BufferInfo)->status = fmi3True;                                   \
        } else {                                                               \
            (BufferInfo)->status = fmi3False;                                  \
        }                                                                      \
    } while (0)
```

Listing 3 : `FMI3_LS_BUS_BUFFER_APPEND` macro

3.9 Modifications of the SIL Kit FMU importer

The **SIL** Kit **FMU** importer was modified to support **CAN** communication in accordance with the **FMI-LS-BUS** network abstraction convention. Under this convention, the `Tx_Data` binary variable may contain a sequence of serialised bus operations rather

than a single **CAN** frame. The **FMI-LS-BUS** implementation guide further specifies that all bus operations associated with a transfer are serialised together within a single **FMI** 3Binary variable. This behaviour is particularly relevant for triggered `Tx_Clock` variables, since transmit data produced during a simulation step is exposed at the subsequent bus communication point [12].

Previously, the importer assumed that each transmit buffer contained a single **CAN** transmit operation. However, the **SIL** Kit importer only supports triggered clocks for **FMI-LS-BUS** communication. Under this communication model, multiple **CAN** transmit operations may accumulate within a single communication step before the bus communication point is reached. The single-frame assumption constituted a defect in the importer implementation.

As a result, when an **FMU** issued multiple **CAN** transmit operations within a single simulation step, the importer handled only the first operation. In addition, the importer allocated memory under the single-frame assumption while copying the entire multi-operation buffer, resulting in heap corruption.

The single-frame assumption is more likely to hold when communication points are scheduled close to individual transmit events, for example, when using countdown clocks or sufficiently small communication step sizes. With triggered transmit clocks, however, the bus communication point is determined by the importer and typically coincides with the end of the communication step.

The modified importer processes the transmitted binary value as a sequence of **FMI-LS-BUS** bus operations. It iterates through the buffer using the operation headers and length fields, identifies **CAN** transmit operations, converts each operation to the corresponding **SIL** Kit **CAN** frame representation, and dispatches the frames individually. This implementation enables the transmission of all **CAN** frames produced during a simulation step at the associated bus communication point, thereby removing the previous single-frame assumption.

Implementation details of the modifications are presented in Appendix A.1.

3.10 Benchmark, scope, and measurement constraints

During evaluation, all runtime measurements use **FMUs** generated from two **vECU** types: a motor **vECU** and a gearbox **vECU**. These two **vECU** types were used in both the **CAN** communication benchmark and the larger participant-count experiments, which included up to 100 **FMU** importer participants. This restriction arose from the limited set of **vECUs** supported by the **FMU** generation pipeline on which the **FMI-LS-BUS** implementation was built.

All runtime results are normalised relative to the corresponding two-participant configuration. The two-participant case represents the smallest benchmark configuration in which both available **vECU** types are present and can exchange **CAN** traffic through the importer. Normalisation, therefore, shows relative scaling with increasing participant count, rather than absolute runtime values.

This benchmark design limits variance in the communication workload. Since participants are replicated from the same two **vECU** configurations, they transmit the same categories of **CAN** frames with similar timing and payload characteristics. The tests, therefore, do not capture the runtime effects of a heterogeneous vehicle network with many distinct ECUs, varying frame periods, distinct **CAN** identifiers, distinct receive filters, and different application-level reactions to incoming traffic.

The limited frame variance also affects receive-side measurements. In the replicated setup, several participants may produce updates for the same logical communication relation during the same synchronisation interval. In such cases, **SIL** Kit may retain only the latest buffered frame value for a given communication point. This may reduce the number of distinct frames received by the **FMU**-side **CAN** handling logic compared with a heterogeneous network, where different **ECUs** own different frames. The benchmark therefore exercises the **CAN** serialisation, forwarding, importer-side deserialisation, and synchronisation mechanisms, but it does not fully represent the receive-side diversity of a complete vehicle network.

The measured values also depend on the selected runtime metric. Several figures report per-process system time for the **FMU** importer participants. This metric measures the CPU time the process spends in kernel mode, excluding time spent waiting to be scheduled. It therefore does not represent the complete elapsed simulation time. Wall-clock measurements are used separately when discussing the total elapsed runtime of the Co-Simulation.

The measurements should be viewed as scalability tests for replicated **vECU** participants rather than comprehensive full-vehicle runtime measurements. However, replication proved useful for this proof of concept, as it increased the number of **FMU** importer processes, **CAN** frames transmitted and received, and **SIL** Kit synchronisation participants, while keeping the workload per participant comparable.

4

Results and Discussion

This chapter presents the results and discussion of the proof-of-concept implementation and evaluation. It describes the implemented **FMI-LS-BUS CAN** interface, the generated **CAN** network mapping, the correctness of frame exchange, the modifications to the **SIL** Kit **FMU** Importer, the computational characteristics of the **CAN** handling implementation, and the measured runtime behaviour.

4.1 Implemented FMI-LS-BUS CAN interface

The prototype exports **vECU** models as Co-Simulation **FMUs** with Low-Cut **FMI-LS-BUS CAN** support. Each generated **FMU** includes the required **FMI-LS-BUS** manifest, **CAN** bus terminals, clock and binary variables, and the runtime structures for the **CAN** bridge.

For each detected **CAN** network, the generated interface exposes one logical **FMI-LS-BUS** bus terminal. The terminal structure follows the clock–binary-variable coupling described in Section 3.4.3. The generated `modelDescription.xml` was therefore sufficient for the selected importer to discover the **CAN** communication variables and perform clocked **FMI-LS-BUS** exchange during event handling.

The runtime support implements the **FMI** functions required by the selected importer for clocked binary communication. In the transmit direction, `fmi3GetClock` and `fmi3GetBinary` expose pending **CAN** operations to the importer. In the receive direction, `fmi3SetClock` and `fmi3SetBinary` receive incoming operations from the importer. The received data is then processed during the discrete-state update phase.

The implemented **CAN** operation support is limited to classical **CAN** Transmit operations. The bridge serialises outgoing **vECU CAN** frames as **FMI-LS-BUS CAN** Transmit operations and deserialises received **CAN** Transmit operations back into the internal **vECU CAN** representation. **CAN-FD**, **CAN-XL**, confirmation operations, arbitration-lost operations, and bus-error operations are not supported by the current prototype.

This demonstrates that **vECUs** may be exported as separate Co-Simulation **FMUs** and that **CAN** frames may be exchanged over **FMI-LS-BUS** without host-configured virtual **CAN** devices. The implemented prototype reliably meets the proof-of-concept

goal by preserving logical **CAN** frame exchange at the level required for the evaluated **vECU** communication scenario.

4.2 Implemented CAN communication flow

The implemented communication flow followed the intended two-phase execution model. Each participant first executed an ordinary **vECU** simulation during `fmi3DoStep`. **CAN** traffic produced during the step was then exchanged during the following **FMI** event-handling phase through the clocked **FMI-LS-BUS** variables.

This separation was observed in the Co-Simulation logs. Outgoing **CAN** frames were staged during the simulation step, exposed through the transmit-side **FMI-LS-BUS** interface at the subsequent communication point, routed by the importer to participants on the same logical **CAN** network, and processed by the receiving **FMUs** during event handling.

The detailed transmit and receive flows are illustrated in Appendix B.

4.3 Automated CAN frame routing

A key outcome of the implementation is that **CAN** communication between **FMUs** can be established without manual signal mapping. Since the generated interface uses **CAN** network names as logical connection points, participants that expose the same **CAN** network can be connected by the importer at the bus level. Data written to one participant's transmit buffer is then routed to the other participants' receive buffers on the same logical network.

Table 1 provides an illustrative example showing how the number of manually mapped signal connections increases with the number of participating **vECUs**. The example assumes that 100 signal connections are required for each participant pair.

Table 1 : Number of signals to be manually mapped as a function of participating vECUs in the Co-Simulation.

vECUs	2	3	4	5	6	7	8	9	10
Signals	100	300	600	1000	1500	2100	2800	3600	4500

This is particularly relevant for larger Co-Simulation setups. Manual signal mapping may be feasible for two participants, but it becomes difficult to inspect and maintain when thousands of connections are required. Bus-level mapping makes the system composition depend primarily on the number of logical **CAN** networks rather than on the number of individual **CAN** signals.

The implemented **FMI-LS-BUS** approach avoids this signal-level mapping problem by moving the connection point to the **CAN** network abstraction. The **FMU** interface is generated from the same **vECU CAN** configuration used by the simulation model, reducing duplication and lowering the risk of inconsistencies among the model, the communication interface, and the simulation setup.

4.4 Communication correctness

The correctness of the implemented communication was evaluated by inspecting the frame transmission and reception behaviour of the corresponding **FMU** participants during Co-Simulation. The goal was to verify that **CAN** frames were routed to the correct participants at the correct emulated time, that the expected frame identifiers were received, and that the transmitted payload data was preserved across the **FMI-LS-BUS** and **SIL** Kit communication path.

Verification was performed by logging the transmitting and receiving **FMU** importer processes. For frame identifiers handled by both participants, the number of transmitted frames was compared with the number of received frames. The logs were also inspected to confirm that the received frame identifiers matched the transmitted identifiers and that the payload bytes were unchanged between transmission and reception.

The results indicated that frames were routed based on shared **CAN** network names and **CAN** frame identifiers. Frames sent by a specific **FMU** on a connected **CAN** network were received by the corresponding participant if that participant had a matching local frame representation. Frames without a matching local representation were ignored by the receiving **FMU**, as expected.

This confirms that the prototype preserves the logical **CAN** communication semantics required for the proof of concept: frames are exported from the transmitting **vECU**, routed through the importer based on **CAN** network metadata, and injected into the receiving **vECU** through the existing **CAN** receive path.

The current correctness result is limited to logical routing and payload preservation. It does not prove that the simulation reproduces physical **CAN** behaviour. In particular, the current prototype does not evaluate arbitration, acknowledgement, retransmission, bus-off behaviour, or timing effects caused by bus bandwidth. If two participants transmit at the same emulated time, the current setup does not resolve the situation in accordance with physical **CAN** arbitration rules. It forwards operations according to the importer behaviour available in the selected toolchain.

This limitation is especially relevant when several replicated **vECU** participants produce frames with the same identifiers. In a physical **CAN** system, ownership of frame identifiers and simultaneous transmission behaviour are constrained by network design and arbitration. In the replicated benchmark, participants generate identical communication patterns across trials. Such a setup is useful for evaluating the importer and **FMU** runtime, but it does not represent a heterogeneous vehicle network in which different **ECUs** own different frames. In addition, the correctness evaluation does not cover preservation of intermediate updates within a communication step.

4.5 Abstraction level of the prototype

The implemented prototype uses the **FMI-LS-BUS** network abstraction, also known as Low-Cut. This abstraction sits below signal-level exchange because it does not expose

decoded physical signal values as separate **FMI** variables. Instead, it exchanges bus operations that represent network-level **CAN** traffic. This aligns with the architecture of a **vECU** that already includes a virtual **CAN** driver and internal **CAN** frame buffers.

The prototype should therefore be understood as a logical frame-exchange implementation, not a complete physical **CAN** bus simulation. It preserves the **CAN** network name, frame identifier, and payload data, and routes frames between participants via the selected importer. It does not model electrical effects, bit-level timing, propagation delay, bus load, arbitration, acknowledgement, retransmission, or bus faults.

This distinction is important because a Low-Cut abstraction level does not automatically imply a detailed bus simulation. Low-cut provides an interface for representing detailed bus behaviour. Whether such behaviour is actually simulated depends on the selected system composition and importer support. In this work, the selected importer routes **CAN** transmission operations between participants but does not provide a full **CAN** bus simulation.

4.6 SIL Kit FMU Importer behaviour

The modified **SIL** Kit **FMU** Importer successfully handled transmit buffers containing multiple concatenated **CAN** operations. This was required because the implemented **FMU** may collect several **CAN** frames during a single simulation step before exposing them through the **FMI-LS-BUS** transmit buffer.

Before the modification, the importer handled only the first **CAN** frame in such a buffer. In addition, copying an entire multi-frame blob into memory allocated for a single frame caused memory corruption. After the modification, the importer parsed the binary transmit buffer as a sequence of **CAN** operations and dispatched each frame individually.

As a result, all frames collected during a single **FMU** step may be transmitted during the subsequent event-handling phase. This behaviour is necessary when using triggered clocks, since several **CAN** frames may become available before the importer reaches the next communication point.

The current importer support is sufficient for forwarding **CAN** Transmit operations, but not for detailed **CAN** bus simulation. In such cases, support for additional operations, such as Confirm, Arbitration Lost, Bus Error, Configuration, and Status, would be required.

For this extension, the existing **FMU** interface could still be used with either a dedicated bus-simulation **FMU** or an alternative importer capable of more detailed simulation, such as **CAN** frame collision handling, arbitration handling, or **CAN** baudrate simulation. The current architecture is compatible with such an extension only if the toolchain supports all relevant **CAN** frame types specified by the **FMI-LS-BUS** standard and if the internal simulation stack processes them correctly.

4.7 Time complexity of CAN communication

Let T denote the computational cost, measured in primitive operations, required to execute the specified procedure. Let c denote the number of **CAN** networks, and let m_i denote the number of **CAN** frames represented on network i. Define

$$m = \max_i m_i.$$

Let p denote the number of payload bytes copied per frame. For classical **CAN**, the payload size satisfies $p \leq 8$, so payload copying is bounded by a small constant. For a fixed generated **FMU**, the number of **CAN** networks, the number of represented frames, and the maximum payload size are fixed. The asymptotic expressions below are therefore mainly useful for describing how the implementation scales with the number of **CAN** frames handled during one communication cycle.

4.7.1 Transmit path

Let n denote the number of transmit calls made by the **vECU CAN** driver during one communication cycle. The transmit path consists of two stages:

- driver-side enqueueing
- bridge-side queue draining and serialisation

For each transmit call on network i, the driver first locates the represented frame by **CAN** identifier. This lookup is implemented as a binary search over the frame set of the network, giving a cost proportional to $\log m_i$. The payload is then copied into the frame buffer, and the frame is inserted into the transmit queue. Thus, one transmit call on network i has cost

$$T_{\text{enqueue op},i} \in O(\log m_i + p).$$

Using $m = \max_i m_i$, the total driver-side enqueueing cost is

$$T_{\text{enqueue}}(n, m, p) \in O(n(\log m + p)).$$

During the bridge step, the transmit queue is drained and converted into **FMI-LS-BUS CAN** transmit operations. Since each queued frame originates from one transmit call, the bridge processes n queued frames during the communication cycle. For each queued frame, the current implementation locates the corresponding **CAN** network by scanning the network list, which costs $O(c)$. Serialising the transmit operation and copying the payload costs $O(p)$.

The implementation also sets the transmission clocks on the relevant networks, which contributes $O(c)$. The bridge-side draining cost is therefore

$$T_{\text{drain}}(c, n, p) \in O(c + n(c + p)).$$

The total transmit-side cost for one communication cycle is

$$T_{\text{tx}}(c, m, n, p) = T_{\text{enqueue}}(n, m, p) + T_{\text{drain}}(c, n, p).$$

Therefore,

$$T_{\text{tx}}(c, m, n, p) \in O(n(\log m + p) + c + n(c + p)).$$

Equivalently,

$$T_{\text{tx}}(c, m, n, p) \in O(c + n(\log m + c + p)).$$

For a fixed generated **FMU** using classical **CAN**, c, m, and p are constants. The transmit-side complexity per communication cycle in terms of handled **CAN** frames therefore reduces to

$$T_{\text{tx}}(n) \in O(n).$$

Thus, for a fixed generated **FMU** using classical **CAN**, the transmit path scales linearly with the number of transmit frames handled during the communication cycle. The cost does not depend on scanning all represented frames, except through the binary-search lookup term, which is constant for a fixed generated **FMU**.

The $O(c)$ term comes from the current implementation, where the bridge locates networks by scanning the network list. This could be reduced by storing a direct network index or reference together with the queued frame.

4.7.2 Receive path

Let n denote the total number of received **CAN** operations processed across all active receive buffers during one communication cycle.

The receive path begins by scanning the c **CAN** networks and checking their receive buffers. This contributes $O(c)$.

The bridge then reads received **FMI-LS-BUS CAN** operations sequentially. Advancing from one operation to the next is constant-time with respect to the number of operations, since each operation header provides the length of the current operation.

For each received operation on network i, the implementation:

- locates the corresponding network by scanning the network list
- locates the represented frame by binary search over the network frame set
- copies the payload into the **vECU** frame buffer
- notifies the **vECU CAN** driver

Thus, one received operation on network i has cost

$$T_{\text{rx op},i} \in O(c + \log m_i + p).$$

Using $m = \max_i m_i$, the total receive-side cost for one communication cycle is

$$T_{\text{rx}}(c, m, n, p) \in O(c + n(c + \log m + p)).$$

For a fixed generated **FMU** using classical **CAN**, c, m, and p are constants. The receive-side complexity per communication cycle in terms of handled **CAN** frames, therefore, reduces to

$$T_{\text{rx}}(n) \in O(n).$$

Thus, for a fixed generated **FMU** using classical **CAN**, the receive path scales linearly with the number of received **CAN** operations handled during the communication cycle. As for the transmit path, the cost does not depend on scanning all represented frames; represented frames are accessed through binary search within the relevant network.

4.8 Runtime impact of CAN volume

The relationship between runtime and **CAN** activity was evaluated by comparing per-process runtime with the number of **CAN** events processed. A **CAN** event denotes a transmitted or received **CAN** frame at a participant. Since each transmitted frame should be delivered to the other participants on the same **CAN** network, the number of **CAN** events grows faster than the number of participants.

For a network with n participants, each participant transmits a fixed number of frames during the simulated interval. Therefore, the total number of transmitted frames grows linearly with the number of participants, giving a simulation-wide transmit complexity of $O(n)$. However, each transmitted frame may be received by up to $n-1$ other participants on the same **CAN** network. The total number of received events, therefore, grows proportionally to $n(n-1)$, giving a simulation-wide receive complexity of $O(n^2)$. As a result, the total number of **CAN** events in the complete simulation also grows approximately quadratically with the number of participants, that is, $O(n^2)$.

Figure 7 shows that per-process runtime increased approximately linearly with the number of **CAN** events each participant handled. This behaviour is important for scalability. Although the total number of **CAN** events grows approximately quadratically over the simulation, they are distributed across importer processes. When enough processor cores are available, participants process their local receive and transmit events in parallel. The elapsed simulation time may therefore scale closer to the per-participant event count than to the simulation-wide event count.

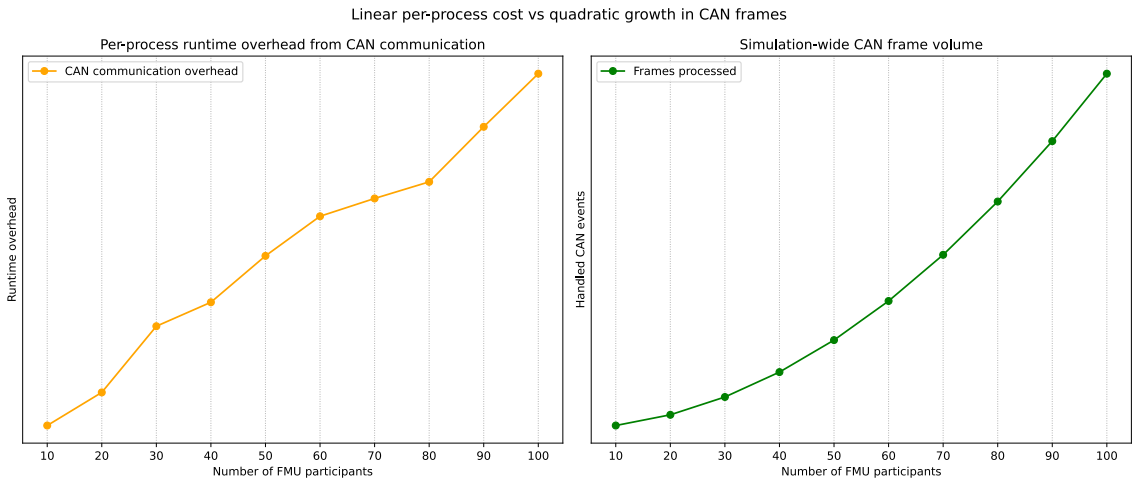


Figure 7 : Per-process runtime versus processed CAN events.

This result illustrates the importance of process-level parallelism in the implementation. The result also motivates the queue-based transmit handling used in the **FMU**-side **CAN** driver. If the driver searched all configured frames during every communication step, the cost would depend on the total number of known frames, even when only a small subset was transmitted. By buffering pending transmit frames, the implementation makes the transmit-side work depend on the number of frames produced during the step. This reduces unnecessary scans and improves scalability for simulations with short step sizes or sparse frame transmission.

4.9 Runtime impact of CAN communication

Runtime measurements were conducted to evaluate the overhead introduced by **CAN** communication. The benchmark comprised two types of communicating **vECUs**. The measurements in Figure 8 were taken on a custom Ubuntu 22.04.5 LTS environment running under WSL2 on an Intel Core i7-13850HX system with 64 GB RAM. The plotted runtime is the per-process system time measured for the **FMU** importer participants.

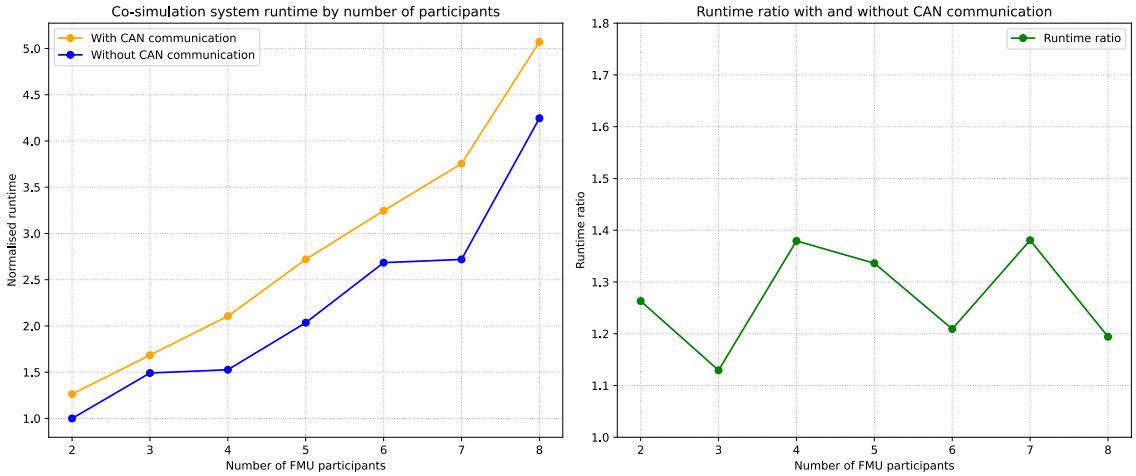


Figure 8 : Co-Simulation system runtime with and without CAN communication.

Enabling **CAN** communication increased the measured runtime compared with the corresponding setup without **CAN** communication. The increase was approximately 25% in the tested configuration. The added work includes queue handling, frame lookup, serialisation, binary buffer transfer, importer-side synchronisation, routing, deserialisation, and injection into the **vECU CAN** receive path. In the tested configuration, this overhead was measurable but not dominant when sufficient processor cores were available. The relative difference between the configurations remained approximately constant as the number of participants increased, indicating that the additional **CAN** communication cost was local to each participant and could therefore be parallelised.

The overhead should not be interpreted as a fixed property of **FMI-LS-BUS**. It depends, for example, on the number of frames transmitted, the number of receivers, the simulation step size, the importer implementation, and the available processor resources.

4.10 Runtime impact of simulation step sizes

In the **SIL** Kit **FMU** Importer setup, the step size determines how much simulated virtual time each participant advances with each call to `fmi3DoStep`.

An experiment evaluated the impact of simulation step size on runtime performance. The step size experiment involved eight participants and increased the step size tenfold, from 1 microsecond to 1 second. The normalised runtime results are shown in Figure 9.

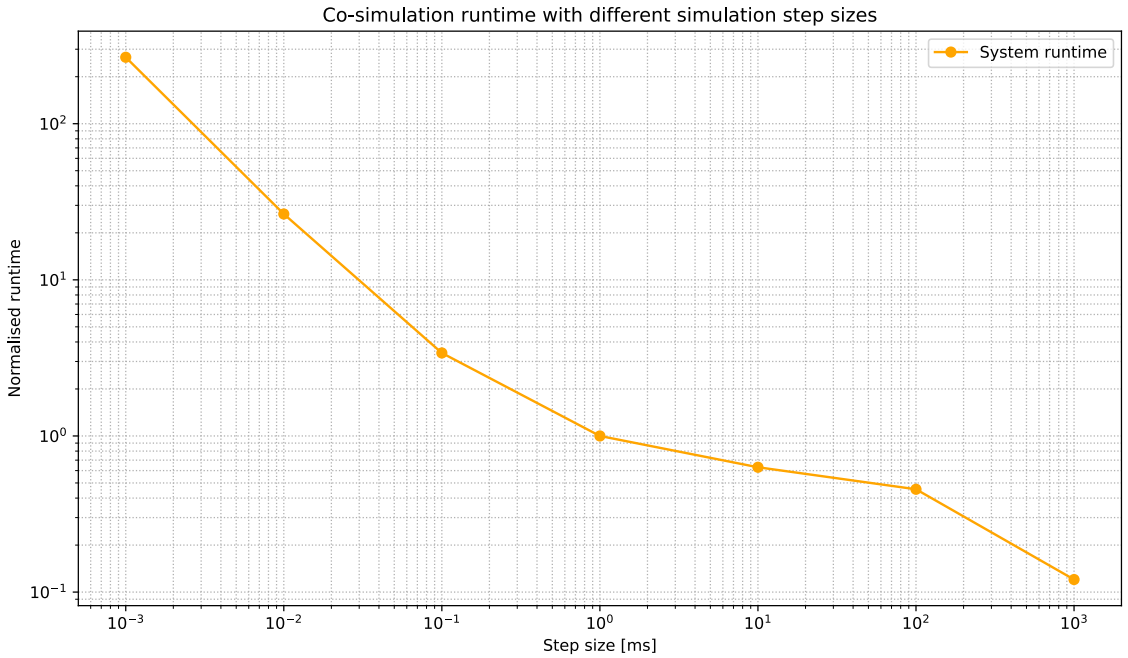


Figure 9 : Runtime per process at different simulation step sizes.

The runtime decreased as the step size increased. A smaller step size requires more `fmi3DoStep` calls to simulate the same virtual time span, which in turn increases the number of **SIL** Kit time-synchronisation points, **FMI** callback sequences, and clocked communication checks. Larger step sizes reduce the number of communication points required to cover a given virtual time span, thereby reducing the associated synchronisation and callback overhead.

The step size also influences the amount of independent work each participant may perform between synchronisations. Smaller step sizes provide finer communication granularity and reduce the time that **CAN** transmit operations remain buffered before the next communication point. Larger step sizes reduce runtime overhead, but they may also bundle more **CAN** transmit operations into each `Tx_Data` transfer.

This trade-off is important for **FMI-LS-BUS CAN** communication with triggered clocks. In this proof-of-concept setup, larger step sizes improved runtime. However, a vehicle-level configuration would need to select a step size that matches the **CAN** communication timing resolution required by the simulation.

4.11 Effect of processor-core availability

A substantial increase in measured runtime was observed when the number of Co-Simulation participants exceeded the number of available processor cores. This effect was most pronounced in the configuration with **CAN** communication enabled, as shown in Figure 10.

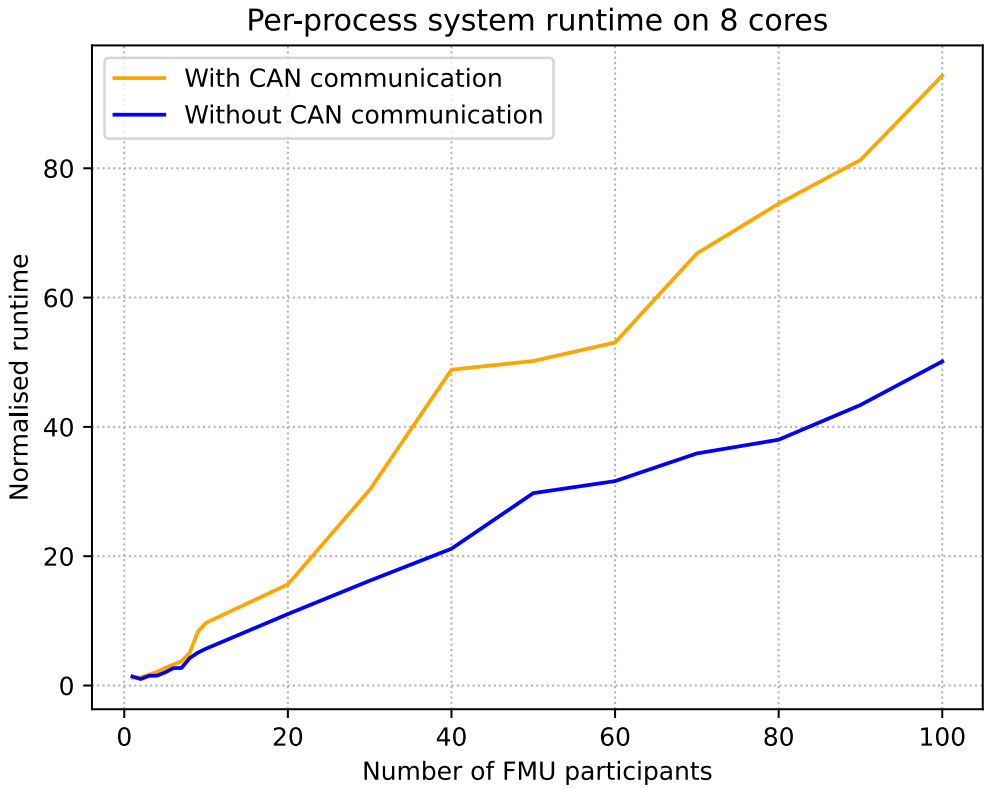


Figure 10 : Per-process runtime by number of participants.

With **CAN** communication disabled and enabled, the measured curves tracked closely up to eight participants. In this range, the processor had enough cores to run the importer processes with limited contention. Beyond eight participants, the curves diverged more rapidly. The configuration with **CAN** communication increased more quickly because each participant performed additional **CAN**-related work at each communication step. When the processes had to share cores, this additional work increased the cost of processor contention.

Additional measurements were conducted with different limits on the number of available processor cores. The results are shown in Figure 11.

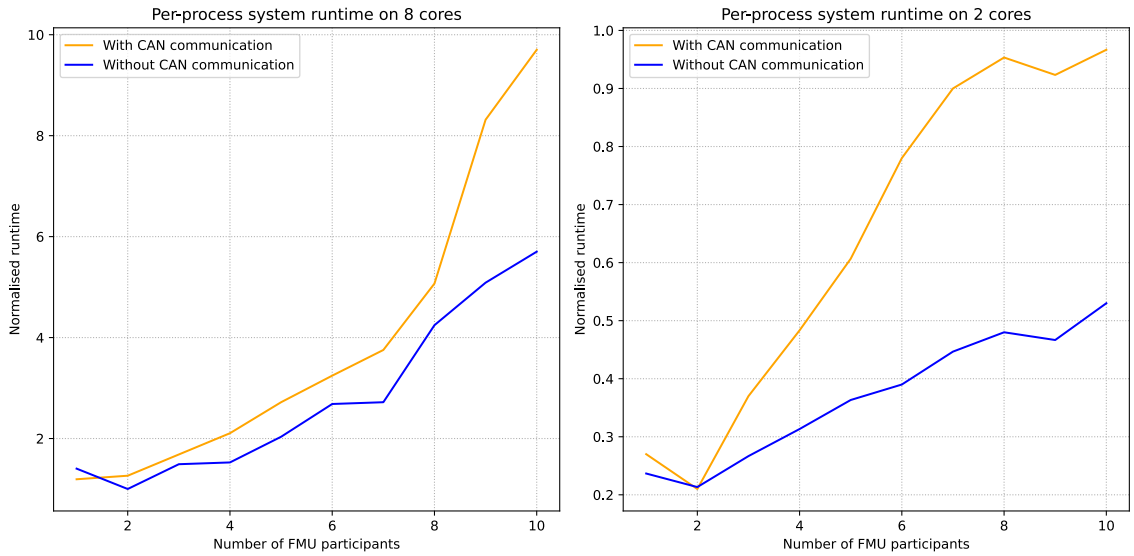


Figure 11 : Effect of processor-core availability on per-process runtime for varying numbers of Co-Simulation participants.

With eight available cores, the per-process runtime remained low while the number of participants was less than or equal to the number of cores. When the participant count approached or exceeded the available core count, the measured per-process system time increased more markedly. With two available cores, the increase occurred at a lower participant count of two. This shows that the scalability of the Co-Simulation depends on both the implementation of **CAN** communication and the ratio of active **FMU** importer processes to available processor cores.

The increase is expected even though per-process system time excludes time spent waiting to be scheduled. When several importer processes share the same cores, the operating system switches between them. Frequent context switches may reduce cache locality, increase cache misses, and increase kernel work related to scheduling and process management. As a result, the processor time spent by each process may increase even when waiting time is not included in the metric.

Figure 12 shows per-process system time for a larger configuration comprising 100 **FMU** importer participants and 32 available cores. The benchmark is conducted on an Intel Xeon Gold 6430 system using a Docker image based on Ubuntu 22.04.5 LTS with 256 GB of RAM.

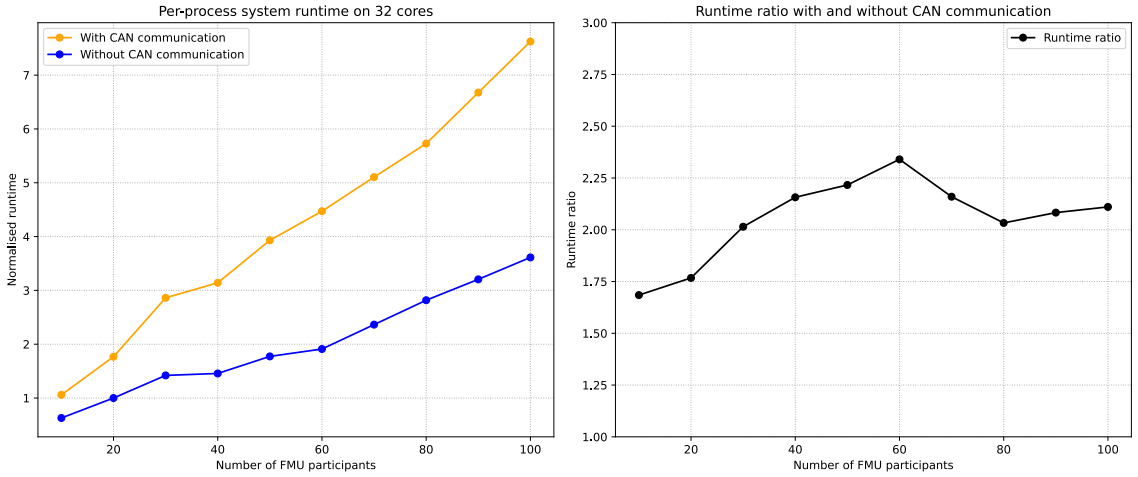


Figure 12 : Per-process system time with 32 available cores.

Wall-clock measurements offer a different view of the same scaling behaviour. Wall-clock time measures the elapsed time for the complete Co-Simulation, including process scheduling, synchronisation, waiting time, and importer-side work. These results are shown in Figure 13.

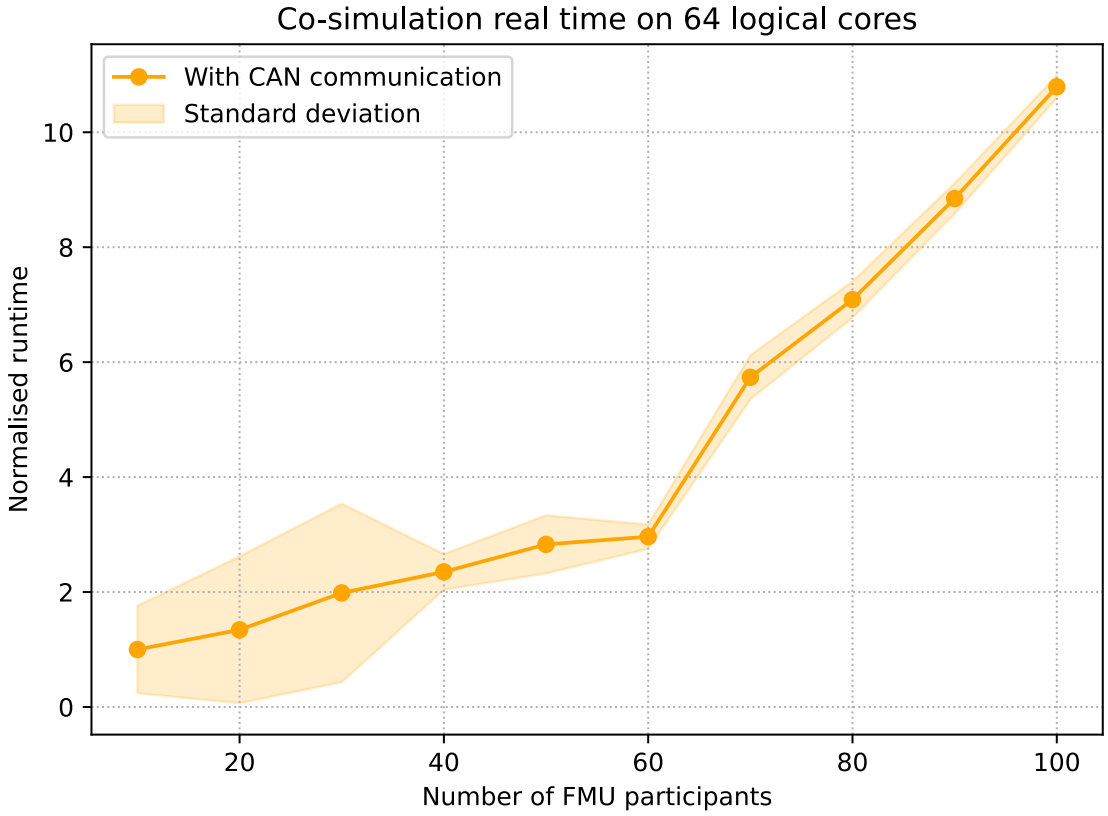


Figure 13 : Wall-clock runtime for Co-Simulation with increasing participant count.

Although the exact runtime will not be disclosed, the Co-Simulation ran substantially faster than real time in the tested case. This indicates that the implementation processed the replicated CAN communication workload faster than real time under the tested conditions.

The wall-clock measurements also show the effect of processor-core availability more directly than per-process system time. When the number of active importer processes exceeds the number of available logical cores, the elapsed runtime includes time spent waiting due to operating-system scheduling. The runtime then increases due to a combination of context switching, reduced cache locality, synchronisation waiting, and scheduling delays.

5

Conclusion

This thesis has presented a proof-of-concept architecture for scalable **vECU** Co-Simulation using **FMI** 3.0 and **FMI-LS-BUS** for **CAN** communication. The implementation shows that **vECUs** can be exported as separate **FMUs** and exchange logical **CAN** frames through generated bus-oriented interfaces.

The results demonstrate that the prototype can preserve frame identifiers and payload data across separately executed participants in the evaluated communication scenario. This provides a basis for scalable software-level **CAN** communication, while the remaining limitations define clear directions for further development.

5.1 Practical implications

The results suggest that **FMI-LS-BUS** is a suitable approach for scalable **vECU** Co-Simulation when the aim is to exchange **CAN** frames between separately executed participants without relying on host-configured virtual **CAN** devices. The approach is particularly promising for workflows that require many simulations to run in parallel and where operating-system-dependent communication resources limit scalability.

The prototype is most directly useful for studies of logical **CAN** communication, integration testing of **vECU** communication paths, and experiments in which frame identifiers and payload data are the relevant correctness criteria. It is less suitable, in its current form, for studies where physical bus timing, arbitration, acknowledgement, error handling, or bandwidth constraints determine the outcome.

The implementation also shows that tool support matters. The standard defines the interface, but the behaviour available in practice depends on importer support. In this thesis, the importer had to be modified to support multi-operation transmit buffers. More advanced use cases would require either a more complete importer implementation or integration with a dedicated bus simulation **FMU**.

5.2 Future work

Several extensions could advance the current proof of concept towards a more complete vehicle-level Co-Simulation platform.

First, **CAN** operation support could be extended beyond classical **CAN** transmit operations. Future versions could support **CAN-FD** and **CAN-XL**, including their larger payloads and additional frame metadata.

Second, future work could investigate integration with a dedicated bus-simulation **FMU** or with an importer that supports bus-simulation. The current prototype evaluates logical frame exchange between network **FMUs** but does not model physical **CAN** bus behaviour. A bus-simulation component could add transmission delay, baud-rate-dependent timing, bandwidth constraints, acknowledgement, error injection, and arbitration between simultaneous transmissions. This would preserve the **FMI-LS-BUS** Low-Cut interface while lowering the effective simulation abstraction level from idealised frame forwarding to more detailed bus behaviour. The **FMU** interface would need to be extended to support bus feedback defined by the **FMI-LS-BUS** standard.

Finally, clock handling could be extended beyond the triggered-clock setup used in this thesis. The selected **SIL** Kit **FMU** importer currently supports **FMI-LS-BUS** **CAN** communication with triggered clocks, where pending transmit data is exposed at the communication point. Future work could evaluate importers or simulation masters that support time-based transmission clocks, in particular, aperiodic countdown clocks. With countdown clocks, an **FMU** can announce a future communication point for buses before the next simulation step. The importer can then adapt the communication step size so that the networked **FMUs** enter event mode at the announced time. This could potentially improve the runtime and performance of the Co-Simulation. However, this requires the **vECU** to reliably determine and expose future transmission times.

Bibliography

[1] T. Burggraf, M. Joswig, M. E. Pfetsch, M. Radons, and S. Ulbrich, 'Semi-automatically optimized calibration of internal combustion engines'. [Online]. Available: https://arxiv.org/abs/1806.10980

[2] S. Borg and V. Hui, 'Multi-Objective Optimization Under the Hood', Master's thesis, 2025. [Online]. Available: https://hdl.handle.net/20.500.12380/310949

[3] 'Controller Area Network (CAN) Protocol Overview'. Accessed: Mar. 23, 2026. [Online]. Available: https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/controller-area-network–can–overview.html

[4] 'CAN Bus Explained - A Simple Intro [2025]'. Accessed: Mar. 25, 2026. [Online]. Available: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial

[5] W. Voss, *A Comprehensible Guide to Controller Area Network*, 2nd edn. Greenfield, MA: Copperhill Technologies Corporation, 2008.

[6] J. Banks, J. S. I. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation*, 5th edn. Harlow, Essex, England: Pearson Education Limited, 2014.

[7] 'Functional Mock-up Interface Specification'. Accessed: Mar. 17, 2026. [Online]. Available: https://fmi-standard.org/docs/main/

[8] 'FMI Layered Standard for Network Communication (FMI-LS-BUS)'. Accessed: Mar. 17, 2026. [Online]. Available: https://modelica.github.io/fmi-ls-bus/main/

[9] 'SIL Kit Overview'. Accessed: Mar. 17, 2026. [Online]. Available: https://vectorgrp.github.io/sil-kit-docs/

[10] 'SIL Kit FMU Importer'. Accessed: Mar. 17, 2026. [Online]. Available: https://github.com/vectorgrp/sil-kit-fmu-importer

[11] 'FMI Tools'. Accessed: Mar. 17, 2026. [Online]. Available: https://fmi-standard.org/tools/

[12] 'FMI Layered Standard for Network Communication (FMI-LS-BUS)'. Accessed: Mar. 17, 2026. [Online]. Available: https://modelica.github.io/fmi-ls-bus/main/#low-cut-tx-rx-data-variables

A

Code

A.1 Implementation of CAN frame stream processing

```C#
private void ProcessOperationStream(uint valref, byte[] data) {
    var offset = 0;

    while (offset < data.Length) {
      var remainingBytes = data.Length - offset;

      // sanity check, make sure frames are even / aligned.
      if (remainingBytes < LsBusOperationHeaderSize) {
        _silKitEntity.Logger.Log(LogLevel.Warn, $"The retrieved CAN operation is malformed. Bytes retrieved: {data}. Remaining bytes {remainingBytes}");
        return;
      }

      var operationSpan = data.AsSpan(offset);
      var operation = (CanOperations)BinaryPrimitives.ReadUInt32LittleEndian(operationSpan
      var operationLength = (int)BinaryPrimitives.ReadUInt32LittleEndian(operationSpan.Slice(4, 4));
      var operationData = operationSpan.Slice(0, operationLength);

      if (operationLength < LsBusOperationHeaderSize || operationLength > remainingBytes) {
        _silKitEntity.Logger.Log(LogLevel.Warn, $"The retrieved CAN data stream is malformed. Not okay operationLength: {operationLength}");
        return;
      }

      switch (operation)
```

```
24      {
25        case CanOperations.Format_Error:
26        {
27          // log the whole operation that caused the error. Data starts
            at the 10th byte
28          var errorData = operationData.Length > 10 ?
            operationData.Slice(10).ToArray() : Array.Empty<byte>();
29          _silKitEntity.Logger.Log(LogLevel.Warn, $"Format Error
            Operation received on Tx_Data with value " +
30            $"reference {valref}. Complete binary data that caused the
              error: " +
31            $"{BitConverter.ToString(errorData)}");
32          break;
33        }
34        case CanOperations.CAN_Transmit:
35        {
36          SendFrame(valref, operationData);
37          break;
38        }
39        case CanOperations.CAN_FD_Transmit:
40        case CanOperations.CAN_XL_Transmit:
41        case CanOperations.Confirm:
42        case CanOperations.Arbitration_Lost:
43        case CanOperations.Bus_Error:
44        case CanOperations.Configuration:
45        case CanOperations.Status:
46        case CanOperations.Wakeup:
47        {
48          // unsupported operation
49          _silKitEntity.Logger.Log(LogLevel.Warn, $"Unsupported
            {operation} Operation received on Tx_Data with value " +
50            $"reference {valref}");
51          break;
52        }
53        default:
54        {
55          // non existing operation
56          _silKitEntity.Logger.Log(LogLevel.Warn, $"Non existing
            Operation received on Tx_Data with value " +
57            $"reference {valref}. Operation code received is:
              {operation}");
58          break;
59        }
60
```

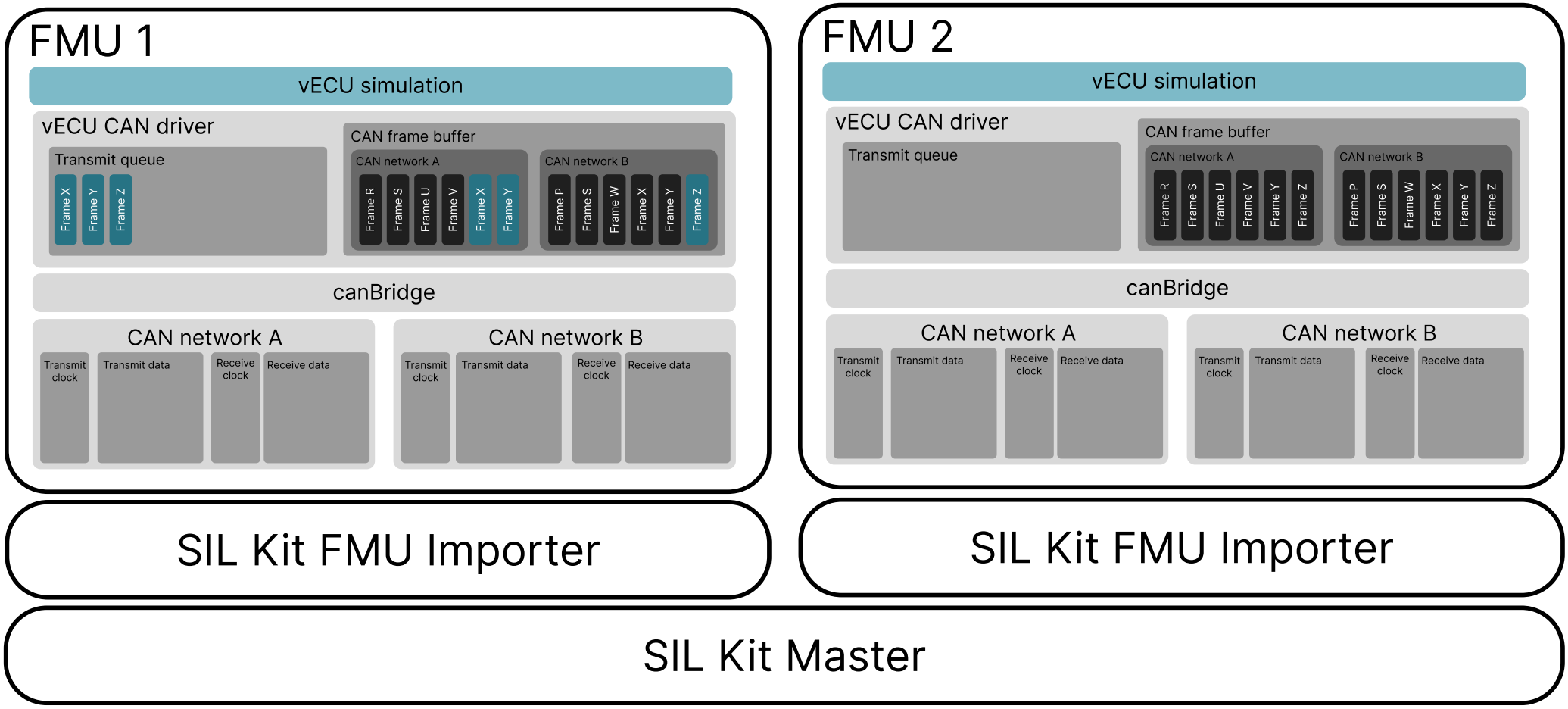
```
61        }
62
63        offset += (int)operationLength; // Move to next operation in blob
64
65      }
66    }
```

B

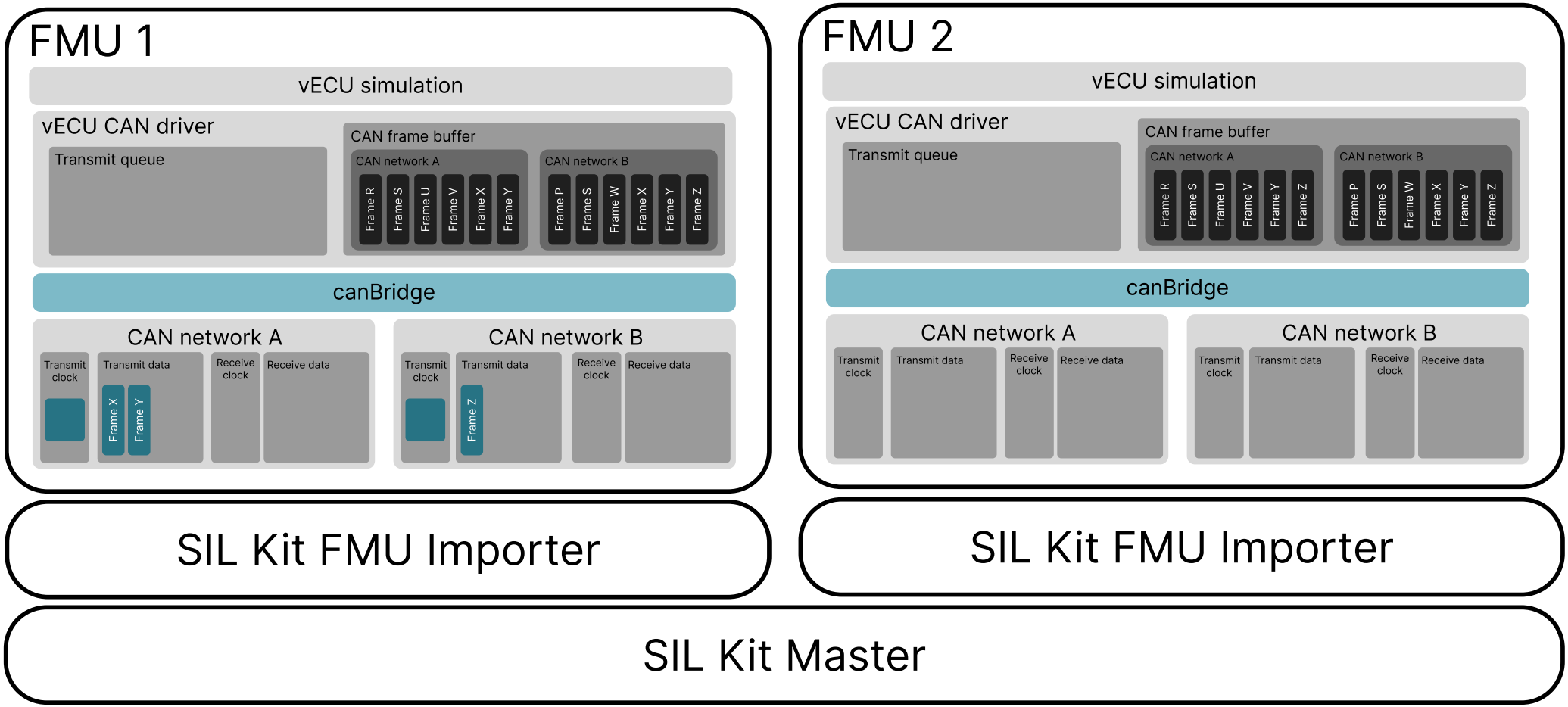
Example CAN Frame Transmission Flow

This example illustrates the transmission of CAN frames X and Y from FMU 1 to FMU 2 over network A, and frame Z to FMU 2 over network B.

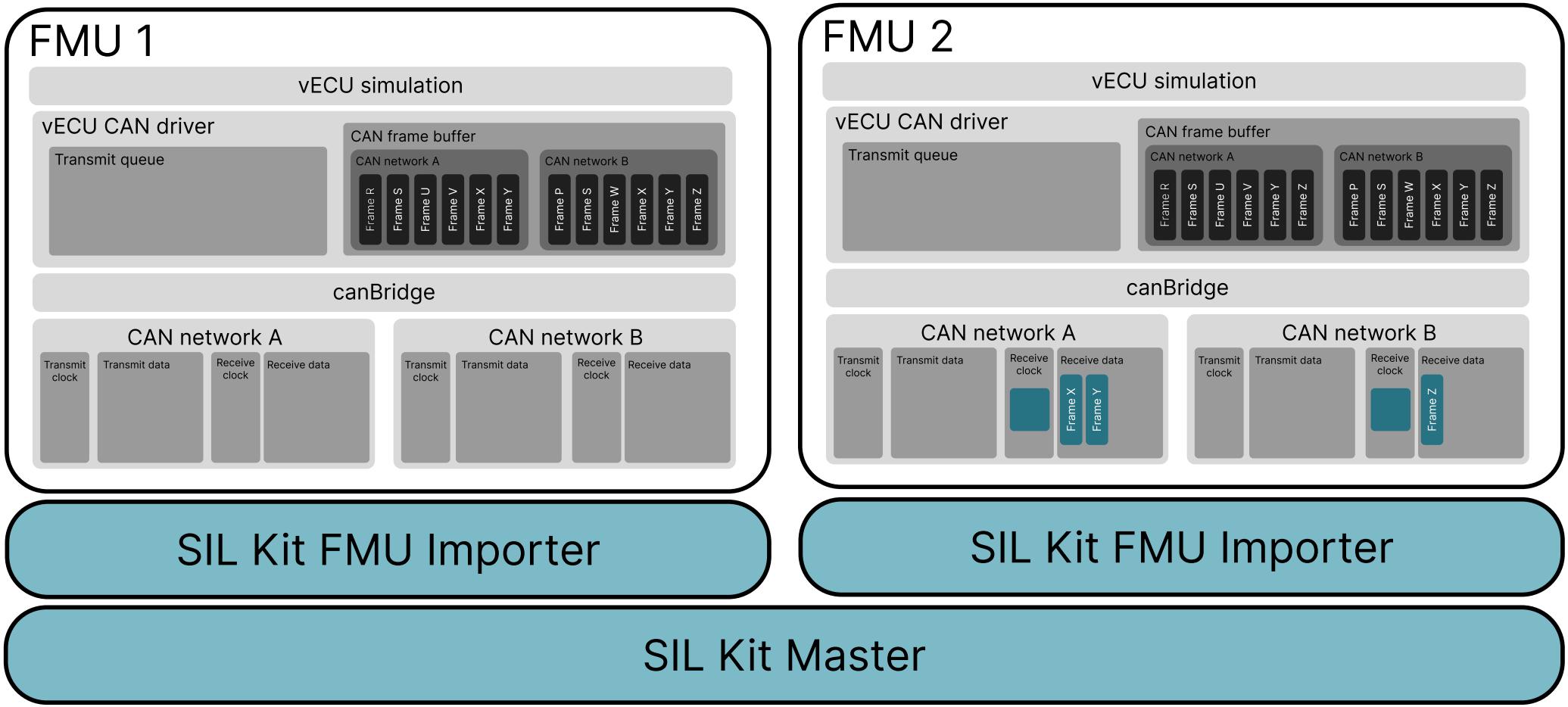
Step 1: Both vECUs simulate one fixed step and update their CAN driver buffers.



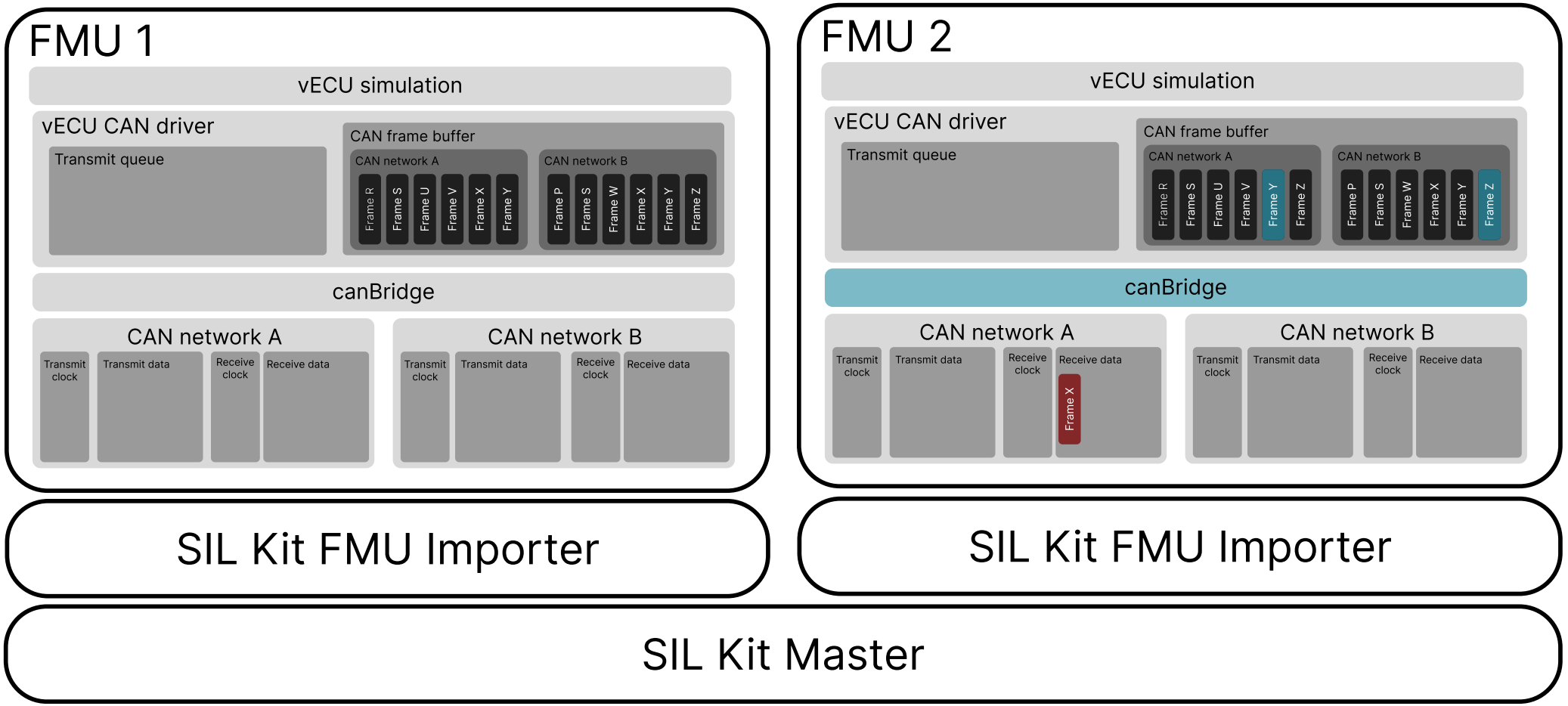
Step 2: The CAN bridge decodes and converts the frames into FMI-type frames, places them in the respective buffers, and sets the corresponding clocks.



Step 3: SIL Kit resets the active transmit clocks, copies the data to the corresponding buffers, and sets the receive clocks.



Step 4: The CAN bridge resets the clocks, copies, decodes, and converts the frames into vECU driver-type frames, and places them in the respective vECU frame buffers.



Step 5: The vECU simulation stack is notified of the received CAN frames.