POLITECNICO DI TORINO
Repository ISTITUZIONALE

Bridging SystemC and QEMU for Automated FMI-Based Virtual Platform Construction

Original
Bridging SystemC and QEMU for Automated FMI-Based Virtual Platform Construction / Albu, A.M., Soldi, D., Autieri, F., Macii, E., Vinco, S.. - In: IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. - ISSN 0278-0070. - (In corso di stampa).

Availability:
This version is available at: 11583/3016057 since: 2026-09-29T11:38:05Z

Publisher:
IEEE

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©9999 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

03 November 2026

Bridging SystemC and QEMU for Automated FMI-Based Virtual Platform Construction

Andrei Mihai Albu [1], Dario Soldi [2], Fabio Autieri [2], Enrico Macii [1], Sara Vinco [1]
[1] Politecnico di Torino, Italy [2] Dumarey Group, Italy
Emails: {name.surname}@polito.it

***Abstract*—Modern automotive systems are evolving into complex cyber-physical systems (CPS), where tightly coupled hardware, software, and physical components must be integrated and validated across multiple abstraction levels. The rise of Software-Defined Vehicles (SDVs) exposes the limitations of existing design flows, which lack scalable and standardized approaches for cross-domain co-simulation. This paper presents an automated framework for virtual platform construction based on the Functional Mock-up Interface (FMI) standard. The framework enables seamless integration of heterogeneous models by encapsulating SystemC designs (spanning RTL, TLM, and AMS) into FMI-compliant Functional Mock-up Units (FMUs) while preserving their execution semantics. A QEMU-based virtual platform is then integrated to support virtual Hardware-in-the-Loop (vHIL), enabling execution of unmodified embedded software in a synchronized co-simulation environment. A unified and automated co-simulation methodology harmonizes discrete-event, continuous-time, and instruction-driven domains through consistent mechanisms for time synchronization, event management, and data exchange. The proposed open-source toolchain improves modularity, scalability, and reproducibility, and is validated on industrial case studies.**

***Index Terms*—Co-Simulation, FMI 3.0, Functional Mock-Up Unit, Hardware-In-The-Loop, QEMU, RISC-V, ARM, RTL, TLM 2.0, AMS, SystemC, Virtual Prototyping**

I. Introduction

The automotive domain represents one of the most challenging cyber-physical system (CPS) scenarios, where hardware, software, and physical components tightly interact under strict safety and reliability constraints [1], [2]. Ensuring correct system behavior requires not only component-level verification but also early and continuous integration across domains.

Virtual platforms have therefore become a key enabler, allowing developers to validate software and system interactions before physical hardware is available. This need is further amplified by the emergence of Software-Defined Vehicles (SDVs), where functionality, performance, and safety-critical features are increasingly software-driven [3]. Scalable and flexible virtual environments are thus required to capture cross-domain interactions from the earliest design stages [4].

The construction of virtual platforms is still a complex task, that requires integrating heterogeneous modeling formalisms and abstraction levels [5]. Even when focusing solely on hardware–software integration, the modeling space spans multiple abstraction levels, including Register Transfer Level (RTL) and Transaction Level Modeling (TLM) for digital hardware, to Analog-Mixed Signal (AMS) for supporting cyber-physical aspects, up to firmware and software, that has to be validated in an Instruction Set Simulator (ISS).

A key limitation in current practice is the *lack of standardized cross-domain co-simulation approaches* capable of integrating heterogeneous domains, execution semantics, and tools within a single virtual platform [6]. Existing solutions often rely on ad-hoc interfaces, manual artifacts, or tightly coupled tool-specific integrations. This fragmentation hinders efficient collaboration and limits the ability to build flexible and modular virtual prototypes, particularly in contexts where traceability and reproducibility are essential.

The *Functional Mock-up Interface (FMI)* standard provides a promising foundation for cross-domain integration [7]. By encapsulating models into *Functional Mock-up Units (FMUs)*, FMI promotes interoperability, IP protection, and integration into heterogeneous tool-chains. Despite these advantages and a wide acceptance for co-simulation in the multi-physics simulation domains [8]–[10], its adoption in the context of hardware/software virtual platforms remains limited [11].

This paper addresses such challenges by proposing a *standardized and automated methodology for virtual platform construction based on FMI*. The main contributions are:

- *Multi-abstraction SystemC encapsulation into FMI*, covering models at RTL, TLM, and AMS levels into FMI 3.0-compliant FMUs, while preserving their discrete-event and/or continuous-time execution semantics;
- *Integration of QEMU for vHIL* by seamless coupling with FMI-based co-simulation, thus enabling execution of unmodified embedded software on virtual hardware synchronized with SystemC models;
- *A unified FMI-based cross-domain co-simulation methodology* that reconciles heterogeneous time semantics across SystemC, QEMU, and FMI, providing well-defined mechanisms for time synchronization, event handling, and data exchange, while decoupling implementation from integration;
- *An automated and open-source toolchain* for generating, integrating, and orchestrating virtual platforms from SystemC descriptions and configuration files. By automatically creating all required integration artifacts, the framework enables a correct-by-construction workflow that improves usability, reproducibility, and reuse while reducing manual effort and integration complexity.

The following of this paper is organized as follows. Sections II and III provide the necessary background and state of the art. Section IV outlines the proposed methodology, that is then applied to case studies in Section V. Section VI discussed the key findings, and Section VII draws our concluding remarks.

II. Background and Semantic Challenges

A. SystemC Simulation Ecosystem

At RTL level, SystemC models describe hardware as concurrent processes triggered by events on signals or channels [12]. Processes are executed following an *evaluation–update cycle*: during evaluation, processes compute new values, that are committed in a later update phase. Deterministic ordering is imposed in case of multiple events occurring at the same simulation time. TLM 2.0 raises abstraction by modeling communication via transactions rather than signals. Components exchange payload objects through initiator/target sockets. SystemC-AMS [13] extends with analog and mixed-signal Models of Computation (MoC). In Timed Data Flow (TDF), modules execute a function periodically with fixed step T_{AMS}. A TDF interface may be used as wrapper for AMS continuous time descriptions: Electrical Linear Network (ELN, conservative) and Linear Signal Flow (LSF, non-conservative) [14]. All SystemC flavors are managed by the same simulation kernel, that converts behaviors specific of each abstraction level into a common event- and time-based semantics.

B. QEMU

QEMU [15] is a dynamic binary translation (DBT) emulator that executes unmodified software on virtual platforms. Guest instructions are translated into host code by the Tiny Code Generator (TCG) and cached as translation blocks. Peripherals are implemented as `SysBusDevices`, and mapped into the guest address space. MMIO accesses invoke device callbacks executed in the QEMU thread context, and interrupts are routed to platform controllers. Execution is driven by an internal loop without a notion of simulation time. For this reason, synchronization with external simulators must be explicitly enforced through the `icount` mechanism, which bounds execution to a fixed number of instructions, and the QMP protocol, which allows external control (e.g., stop).

C. FMI 3.0

FMI standardizes co-simulation of heterogeneous models [7]. Each model is packaged as a FMU, containing an XML-based model description file (`modelDescription.xml`) that defines variables, parameters, input/output signals, data types, etc., plus compiled libraries implementing the standard FMI API. FMUs execute according to three different solutions. The focus of this paper is on the co-simulation integration style, where each FMU advances independently over communication steps of size Δt, defined by a master. The typical execution loop consists of setting inputs (`fmi3SetXXX`, where `XXX` is one of the FMI data types), advancing time (`fmi3DoStep`), and retrieving outputs (`fmi3GetXXX`). Variables are accessed through value references (VRs), ensuring typed and consistent data exchange.

D. Motivation and Semantic Mismatches

Integrating SystemC, QEMU, and FMI introduces fundamental mismatches in execution semantics, scheduling, and event handling.

Execution model: SystemC relies on an event-driven simulation with delta cycles (RTL/TLM) or fixed-step iteration (TDF), QEMU executes instructions sequentially without an intrinsic notion of time, and FMI enforces a step-based progression controlled by an external master. These models are inherently incompatible, requiring each `fmi3DoStep` call to be mapped onto a corresponding execution fragment.

Scheduling: Each domain is governed by an independent scheduler. Correct integration requires well-defined synchronization points at which all domains yield control.

Event granularity: Hardware events (e.g., interrupts) may occur at finer temporal resolution than the FMI communication step. Preserving causality requires that all events within a time interval are either resolved or safely deferred.

Determinism: SystemC models are inherently deterministic, whereas QEMU execution may exhibit non-determinism. Since FMI does not enforce determinism, additional control mechanisms are required to ensure reproducible execution.

These mismatches cannot be resolved at the interface level alone and require explicit coordination mechanisms.

III. Related Work

Table I summarizes prior research.

TABLE I
Comparison of related works, listing whether the source is left unchanged (Non-intr.), automation (Auto.), open-source release (Open), and co-simulation support. (Multi-dom.).

Ref.	FMI	SystemC			QEMU	Non-intr.	Auto.	Open	Multi-dom.
		RTL	TLM	AMS					
SystemC-FMI Approaches									
[16]	1.0	✓	✗	✗	✗	✓	✗	✗	✓
[17]	2.0	✓	✗	✗	✗	✓	✗	✗	✓
[18]	2.0	✓	✓	✗	✗	✗	✓	✗	✓
[19]	2.0	✗	✓	✗	✗	✓	✗	✗	✓
[20]	2.0	✗	✓	✗	✗	✓	✗	✗	✓
[21]	2.0	✗	✓	✗	✗	✓	✗	✗	✓
[22]	2.0	✗	✓	✗	✗	✓	✓	✗	✓
[23]	2.0	✗	✓	✗	✗	✗	✗	✓	✓
[24]	3.0	✓	✗	✗	✗	✓	✓	✓	✓
[25]	3.0	✗	✓	✗	✗	✓	✓	✓	✓
Virtual Platform Approaches									
[26]	–	✗	✗	✗	✓	–	✗	✗	✗
[27]	–	✗	✗	✗	✓	–	✗	✗	✗
[28]	–	✗	✗	✗	✓	–	✗	✗	✗
[29]	–	✗	✗	✗	✓	–	✗	✗	✗
[30]	–	✗	✓	✗	✓	✓	✗	✗	✗
[31]	–	✗	✓	✗	✓	–	✓	✓	✗
[32]	–	✗	✓	✗	✓	–	✓	✓	✗
Orchestration Approaches									
[33]	2.0	✗	✗	✗	✗	–	✓	✓	✓
[34]	2.0	✗	✗	✗	✗	–	✗	✗	✓
[35]	2.0	✗	✗	✗	✗	–	✗	✗	✓
[36]	2.0	✗	✗	✗	✗	–	✗	✗	✓
[37]	3.0	✗	✗	✗	✗	–	✗	✗	✓
This work	**3.0**	✓	✓	✓	✓	✓	✓	✓	✓

A. SystemC–FMI Integration

Early works integrating SystemC models within FMI-based co-simulation frameworks focused on wrapping RTL-level SystemC models as FMI Co-Simulation FMUs [17], [24], demonstrating correct time advancement but limited to signal-level designs. Subsequent efforts extended support to TLM-based models [18], [25], enabling transaction-level integration. In parallel, several works targeted virtual ECU co-simulation

and orchestration [16], [19], focusing on interface standardization rather than preserving full SystemC semantics across abstraction levels. While these approaches demonstrate the potential of FMI-based integration for SystemC models, they typically focus on specific abstraction levels or individual aspects of the integration flow.

More recent solutions improve automation and usability, either through proprietary toolchains [20], [21] or reusable component libraries [38]. However, these approaches remain limited in scope, typically supporting only specific abstraction levels (e.g., TLM), requiring intrusive modifications, or restricting interoperability across tools.

Despite these advances, existing approaches do not simultaneously address (i) multi-abstraction support, (ii) preservation of execution semantics, and (iii) automated generation of integration artifacts. Furthermore, none of these approaches simultaneously supports multi-abstraction SystemC integration, full-system virtual platform execution, and standardized orchestration within a unified framework.

B. QEMU-Based Virtual Platforms

QEMU [15] is widely adopted as an ISS in hardware/software co-simulation platforms. Existing integration strategies can be broadly categorized into two approaches. In *cross-process coupling*, QEMU executes independently from SystemC, with shared memory or IPC mechanisms coordinating data exchange [26]–[29]. Although this strategy improves timing fidelity, it remains platform-specific and lacks standardization. In case of *tight SystemC embedding*, QEMU is encapsulated as a SystemC module with TLM 2.0 initiator sockets [30]. This reduces IPC overhead and improves timing accuracy, but tightly couples the platform to SystemC, limiting flexibility and reuse. Container-based deployments [31], [32] enhance scalability and reproducibility, yet still rely on bespoke SystemC-centric integration. In both cases, no standardized interface exists for integrating external or heterogeneous models, nor for decoupling hardware descriptions from SystemC-specific implementations.

C. Multi-Domain Co-Simulation Orchestration

System-level co-simulation must reconcile heterogeneous notions of time, couple different solvers, and handle multi-abstraction models across different domains. Existing frameworks address subsets of this problem, but remain limited in scope. Model orchestration frameworks, such as OMSimulator [33], support FMI-based composition (e.g., FMI 2.0 and SSP), but lack native SystemC integration and cannot represent transaction-level or ISS interactions. Automotive Software-in-the-Loop (SiL) frameworks [37] leverage FMI 3.0 to model software timing behavior, yet hardware is typically represented at a purely functional level, without SystemC fidelity or integration with virtual platforms. Signal-level co-simulation approaches [34], [35] focus on communication protocols and interface generation, but do not capture structural hardware models or multi-abstraction SystemC descriptions. Finally, formal time-coupling methods [36] provide theoretical foundations for bridging discrete-event and continuous-time semantics, but do not offer practical integration with SystemC models or instruction-set simulators.

As a result, existing approaches either operate at continuous-time or functional signal levels without SystemC awareness, or target specific aspects of co-simulation without supporting full-system virtual platforms. No prior work combines multi-abstraction SystemC integration, QEMU-based virtual platform execution, and FMI 3.0-compliant orchestration within a single non-intrusive and automated framework.

IV. Methodology

The proposed framework introduces a unified and automated integration methodology for heterogeneous hardware/software co-simulation. The design is guided by four key principles: (i) preservation of native execution semantics across domains, (ii) non-intrusive and correct integration of existing models, (iii) standardized orchestration through FMI, and (iv) automation of virtual platform construction to improve usability, reproducibility, and workflow efficiency.

The framework combines automated SystemC-to-FMU encapsulation with a QEMU-based virtual platform, coordinated through a unified co-simulation layer. As illustrated in Figure 1, the framework is organized into four main stages:

- *Model analysis:* the SystemC design and a configuration file are parsed to extract structural and semantic information, and to generate the `modelDescription.xml` defining the FMU interface (Section IV-A);
- *SystemC-to-FMU wrapping*: the SystemC model is automatically encapsulated into a FMU executable, including all required wrapper logic (Section IV-B);
- *QEMU setup and software construction*: the QEMU-based virtual platform is configured, and a corresponding software stack (including device drivers and a baseline application) is generated. The FMU is integrated as a virtual device (Section IV-C);
- *Virtual platform execution*: the resulting system enables full-system vHIL execution, allowing unmodified embedded software to interact with the simulated hardware (Section IV-D).

The whole process is automated, and is made available at [39].

A. Model Analysis

The first step retrieves necessary information from the SystemC initial design and from an input configuration file, to *map SystemC ports and payloads to FMI variables*. The SystemC source file tree is parsed to identify the top-level module, that is then analysed by a regular-expression pass to extract the complete port list and to identify the level of abstraction (RTL, TLM, or AMS). For each RTL or TDF port, the parser collects identifier, direction, and datatype (together with its bit-width and signedness). For TLM 2.0 designs, the parser detects the presence of sockets and extracts the payload fields (e.g., address and response status). Each field is treated separately, similarly to RTL ports, as FMI lacks a native transaction abstraction. For AMS TDF designs, the parser also identifies the fixed activation timestep of the design and/or its ports, that is used at later stages.

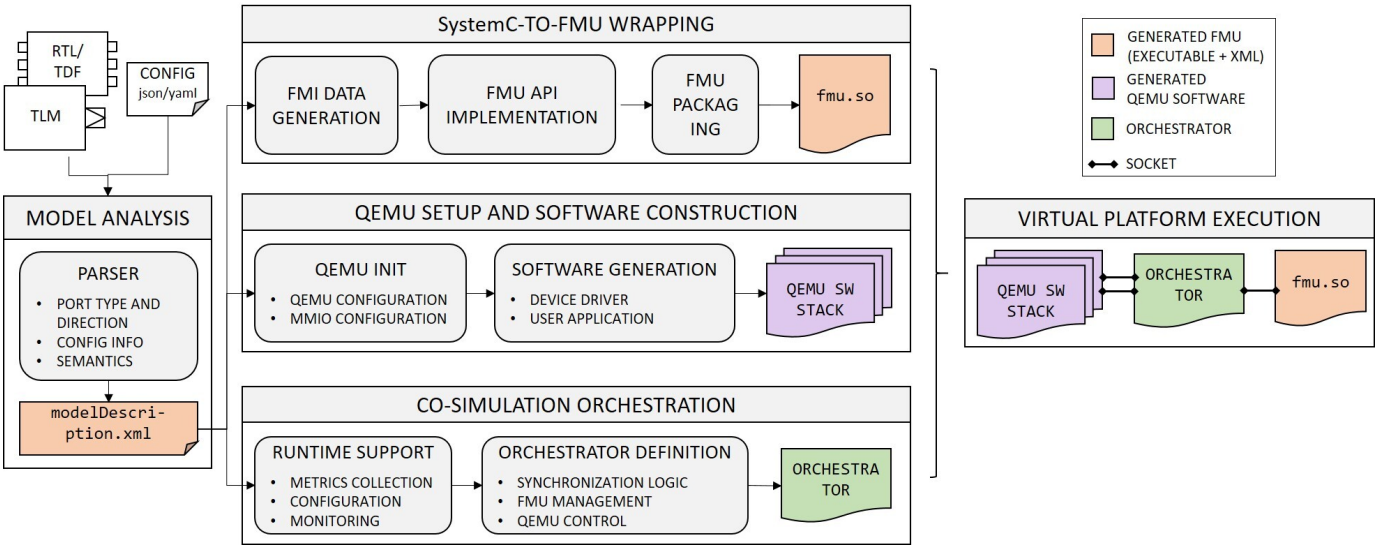


Fig. 1. Overview of the SystemC-to-FMU toolchain and vHIL integration, highlighting the role of the orchestration layer in coordinating QEMU, FMU, and SystemC. Starting from the initial SystemC description and a configuration file, model analysis (left) drives automated FMU generation (top) and QEMU-based platform construction (middle). An orchestrator is then defined (bottom) to combine FMU and QEMU at runtime for vHIL execution.

The current implementation relies on lightweight source-code analysis, which proved sufficient across all evaluated case studies. Since the proposed methodology only requires interface-level information, alternative front-ends based on abstract syntax tree (AST) analysis [40] can be adopted without affecting the remainder of the flow.

The corresponding FMI variables are then generated, by using the same identifier as in the initial description, and by mapping SystemC and C++ data types to FMI 3.0 data types. The mapping in Table II is exhaustive and conservative: signedness is preserved end-to-end, with signed and unsigned types unchanged throughout the toolchain. Type consistency is enforced across all layers.

TABLE II
SYSTEMC-FMI 3.0 DATA-TYPE MAPPING.

SystemC / C++	FMI 3.0
`bool`, `sc_logic`	`fmi3Boolean`
`sc_bv<N>`, `sc_lv<N>`	`fmi3Binary`
`sc_int<N>`, `int*_t`	`fmi3Int8-Int64`
`sc_uint<N>`, `uint*_t`	`fmi3UInt8-UInt64`
`float`	`fmi3Float32`
`double`	`fmi3Float64`
`char*`, `std::string`	`fmi3String`
`sc_fixed`, `sc_ufixed`	`fmi3Float64`
`tlm_generic_payload`	*decomposed*

The second step is then to retrieve *configuration information*. The input YAML/JSON configuration file contains information related to user defined settings, including the processor to be emulated in the ISS, paths to the hard disk image and operating system version. Optional parameters (e.g., interrupt number, MMIO base address and range, socket addresses and ports, co-simulation timestep size) may be assigned default values. This information is saved in an intermediate representation (IR), that is passed on to the next steps.

B. SystemC-to-FMU wrapping

This phase encapsulates the SystemC model into an FMI-compliant execution unit by generating a wrapper that bridges FMI APIs with native SystemC execution semantics.

The output is an executable that, paired with the formerly generated `modelDescription.xml` file, composes a standard-compliant FMU.

1) FMI data generation: The first ingredient of the wrapper is a `struct` collecting relevant information:

- one shadow variable per mapped port (or payload field) with the corresponding FMI type;
- pointers to all heap-allocated SystemC modules and interconnecting signals;
- auxiliary control fields derived from the configuration file (e.g., flags to handle interrupts or halt conditions).

2) FMU API Implementation: The next step involves implementing the prototype functions required by the FMI standard. The main functions are the following:

- `fmi3InstantiateCoSimulation`: allocates the state object, and constructs all SystemC modules on the heap. It also allocates RTL and AMS signals and TLM sockets, that are bound to SystemC ports or sockets[1]. VCD/waveform tracing is initialized at elaboration time to ensure coverage across all subsequent activations of SystemC. No simulation is started yet.
- `fmi3EnterInitializationMode` / `fmi3ExitInitializationMode`: the enter function initializes all shadow variables to design-specific safe defaults. The exit function propagates initial values provided by the master into the SystemC domain and enforces abstraction-specific consistency.
- `fmi3SetXXX` / `fmi3GetXXX`: transfers data between FMI shadow variables and SystemC ports or payload

[1]Note that all top-level entity ports and sockets must be bound to either ports or signals to allow successful SystemC simulation.

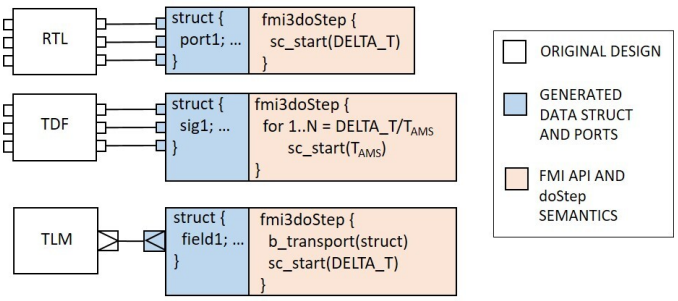


Fig. 2. **Wrapping strategy at RTL, TDF, and TLM abstraction levels. Port aggregation structs are auto-generated per model, and `fmi3DoStep` semantics are adapted to each abstraction's timing model.**

fields. Abstraction-specific write semantics are detailed in the following.

- `fmi3DoStep`: advances SystemC simulation by invoking `sc_start(communicationStepSize)`, where `sc_start` is the function that activates SystemC simulation by a given delta time (`communicationStepSize`). The function tracks also the latest time reached by the SystemC domain through a `lastSuccessfulTime` variable. The semantics of this function strictly depends on the level of abstraction of the SystemC design, and is detailed in the following subsection.
- `fmi3EnterEventMode`: invoked upon event conditions (e.g., interrupts), enabling event-driven synchronization between the FMU and the master algorithm.
- `fmi3FreeInstance`: finalizes the instance by closing trace files, deleting all SystemC modules and signals in reverse construction order, and deallocating the struct.

a) `fmi3DoStep` semantics: The implementation of the `fmi3DoStep` function strictly depends on the abstraction level of the target SystemC design (Figure 2). The behavior is the same for any SystemC design: to advance simulation time by Δt through an invocation of the `sc_start` primitive. The management of this step reflects the abstraction level of the SystemC design to be wrapped:

- At RTL, an invocation of the `fmi3DoStep` function corresponds to an invocation of the `sc_start(`Δt`)`, that allows to advance simulation time, signal propagation and process activation by the given time.
- At AMS TDF level, the TDF activation time T_{AMS} may differ from the FMI time step Δt. It is thus necessary to define a wrapper that updates TDF input ports with period T_{AMS}, and that extrapolates the value and map it onto a pure C++ variable.
  Upon invocation of a `fmi3DoStep`, time thus is advanced by invoking `sc_start(`NT_{AMS}`)` with $N = \text{round}(\Delta t / T_{\text{AMS}})$, and outputs are read from buffered values. This ensures time compatibility between the AMS and the FMI domain.
- In TLM 2.0, `fmi3Set`$\star$ updates the payload with FMI-provided values. The transaction is executed via transport primitives followed by `sc_start(`Δt`)`, after which responses are latched; events beyond Δt are not observable. The TLM socket is instantiated in *fmi3InstantiateCoSimulation*. A key limitation is that events scheduled *beyond* Δt (e.g., via threads or deferred `wait()`) are discarded due to the time-bounded `sc_start()`. This follows from the FMI co-simulation contract, which forbids advancing time past the next communication point. Thus, all observable side effects must complete within the transport call.

This translates into an advancement of simulation time by Δt in all configurations: the different management strategies are necessary to map the time advancements onto the specific mechanisms of each abstraction level (e.g., port/payload data management, TDF activation time compatibility).

3) FMU packaging: The result is a complete and self-contained FMU, including the `modelDescription.xml` file and a standalone executable file, obtained by compiling the SystemC design and the generated FMI wrapper.

4) Achieving full SystemC-FMI compatibility: This section overviews how SystemC constructs or mechanisms not easily supported by FMI are managed by the proposed solution.

a) Handling `sc_stop()` invocations: Some RTL designs use `sc_stop()` to terminate the simulation upon critical conditions. Since the FMI master may continue calling `fmi3DoStep` after the SystemC kernel has stopped, this can trigger error E546 (`sc_start` after `sc_stop`) and abort the simulation. To prevent this, the wrapper maintains a boolean `stopped` flag. When E546 is detected, the flag is set, and subsequent `fmi3DoStep` calls return immediately, thus terminating the simulation in a controlled way.

b) Clock management: In conventional RTL simulation the `sc_clock` is declared in `sc_main()`. In the current FMU-wrapped code structure, the `sc_main()` entry point is never reached, as the FMI C API functions are invoked directly by the master. The clock signal must thus be managed entirely within the wrapper. The generator instantiates an `sc_clock` object inside FMI data struct, by using initial configuration provided in the input configuration file (i.e., period, duty cycle, and initial phase). The clock is deliberately *not* exposed as an FMI variable, as this would create too frequent synchronization samples (clock period is much finer than the FMI Δt), while FMI co-simulation requires a *functional* aggregate response of the RTL over a time interval, not edge-level control. To handle the clock, each `fmi3DoStep` with step size Δt calls `sc_start(`Δt`)`. During this interval the internal `sc_clock` toggles $\lfloor \Delta t / (T_{\text{clk}}/2) \rfloor$ times, advancing all registered RTL processes by the corresponding number of rising and falling edges.

c) Support for master-driven `wait()`: To model latencies and timing behaviors, the master may request a timed `wait()` that advances SystemC time without new inputs. To support this, a dedicated module enables the simulation to progress for a specified interval while only internal events are processed. This behavior is controlled by a `wait_requested` flag in the FMI data structure. When set, `fmi3DoStep` advances the simulation via `sc_start` without updating any of the design inputs.

C. QEMU Setup and Software Construction

The software side of the virtual platform is constructed by automatically generating a QEMU-based execution environment. The generated software components provide the interface governing the interaction of unmodified embedded software with the FMU-backed model as if it were a physical hardware peripheral.

1) QEMU initialization and software deployment: The QEMU instance is automatically launched by using the configuration information extracted from the initial YAML/JSON file (including kernel image, root filesystem, MMIO and interrupt settings, and ISS version). QEMU is configured in instruction-counting mode (`-icount`) to ensure deterministic execution and enable estimation of the QEMU internal time (i.e., the length of simulated execution), which is required for synchronization with the FMU `fmi3DoStep` function. QMP is enabled to allow external control of QEMU (e.g., pause, resume, run for a given number of instructions). Once the virtual machine is booted, the generated software stack (driver and application) is transferred into the guest, compiled, and dynamically loaded. During initialization, the driver establishes the communication channel with the host-side co-simulation master, enabling synchronized interaction between software execution and hardware model evaluation.

2) QEMU MMIO as Interface to the Virtual Platform: The interaction between guest software and the FMU-backed model is realized through a memory-mapped virtual device and an interrupt-driven handshake mechanism. This solution enables transparent interaction between software and hardware models without modifying the software stack.

The virtual device is implemented as a QEMU `SysBusDevice` and exposes a register file where each FMI variable is mapped to a memory-mapped location. Input registers are written by the guest application, while output registers are updated by the host after FMU evaluation. This design provides a transparent abstraction in which software interacts with the FMU as with a conventional hardware peripheral.

Co-simulation is coordinated through an interrupt-based protocol. A dedicated register (handled as an interrupt) is used to notify that the guest has completed writing input data and that the host can proceed with FMU evaluation. This avoids polling-based synchronization and ensures a well-defined interaction boundary between software execution and model simulation. QEMU includes also additional classes, including a `MetricsCollector` used to sample relevant performance metrics (e.g., memory, timers).

The resulting execution flow is as follows. The guest writes the input registers and triggers the transaction via the dedicated control register. The corresponding MMIO handler schedules an interrupt in the QEMU event loop, which is delivered to the guest through the platform interrupt controller. The interrupt service routine schedules deferred processing in the driver, enabling the request to be forwarded to the host. After FMU evaluation, the response is written back to the MMIO output registers, and the interrupt is cleared when the guest reads the result, re-arming the system for the next transaction.

3) Device driver implementation: The device driver provides the software bridge between the guest application and the host-side co-simulation infrastructure, exposing the virtual device as a standard memory-mapped blocking device.

The driver binds to the virtual device via a Device Tree entry, maps the MMIO region, registers an interrupt handler, establishes a kernel-space communication channel with the host, and exposes a `miscdevice` interface to user space. The driver is structured in two layers: a lightweight interrupt service routine (ISR) and a deferred execution stage implemented through an ordered workqueue. The ISR handles interrupts with minimal latency by scheduling a work item and immediately returning. The workqueue executes the full transaction: reading input values from MMIO, serializing them into a request structure, sending them to the host, receiving the response, writing output values back to MMIO, and finally waking any blocked user-space process. The use of an ordered workqueue is essential to ensure correctness: it serializes all transactions, preventing concurrent interrupt handling from interleaving partial request/response exchanges and guaranteeing a one-to-one correspondence between software transactions and FMU evaluation steps.

Communication between the driver and the host is implemented through a binary wire protocol based on packed structures. Data is transmitted in little-endian format with strict type consistency across kernel, user space, and host-side representations, ensuring correct interpretation of values and preventing silent data corruption.

4) User app skeleton generation: A simple user application is generated, writing default values to the FMU and reading back the corresponding outputs. This is necessary to validate the constructed simulation and the virtual device mechanism, and provides a starting point for future custom implementation.

D. Co-simulation Orchestration

This section introduces a co-simulation orchestration layer that provides a unified execution model across SystemC, QEMU, and FMI to enable correct-by-construction integration of the heterogeneous execution domains. While Section IV-B2a defines how each FMU advances locally through abstraction-specific `fmi3DoStep` semantics, a global orchestration layer is indeed required to synchronize all domains consistently. The key idea is to enforce a global lock-step execution model (represented in Figure 3), in which all domains advance synchronously through FMI communication steps, and to address the semantic mismatches.

To achieve this, the proposed framework introduces an orchestration layer, which acts as the central coordinator of the co-simulation process. Its role is to *align the instruction-driven execution of QEMU with the step-based progression of FMI*, while preserving causality, timing consistency, and determinism. This separation between local execution semantics and global orchestration enables modular integration of heterogeneous models while preserving their native behavior.

1) Communication setup: As depicted on the right-hand side of Figure 1, communication is realized through dedicated sockets separating data exchange and control signaling. QEMU-side communication requires two sockets: one to

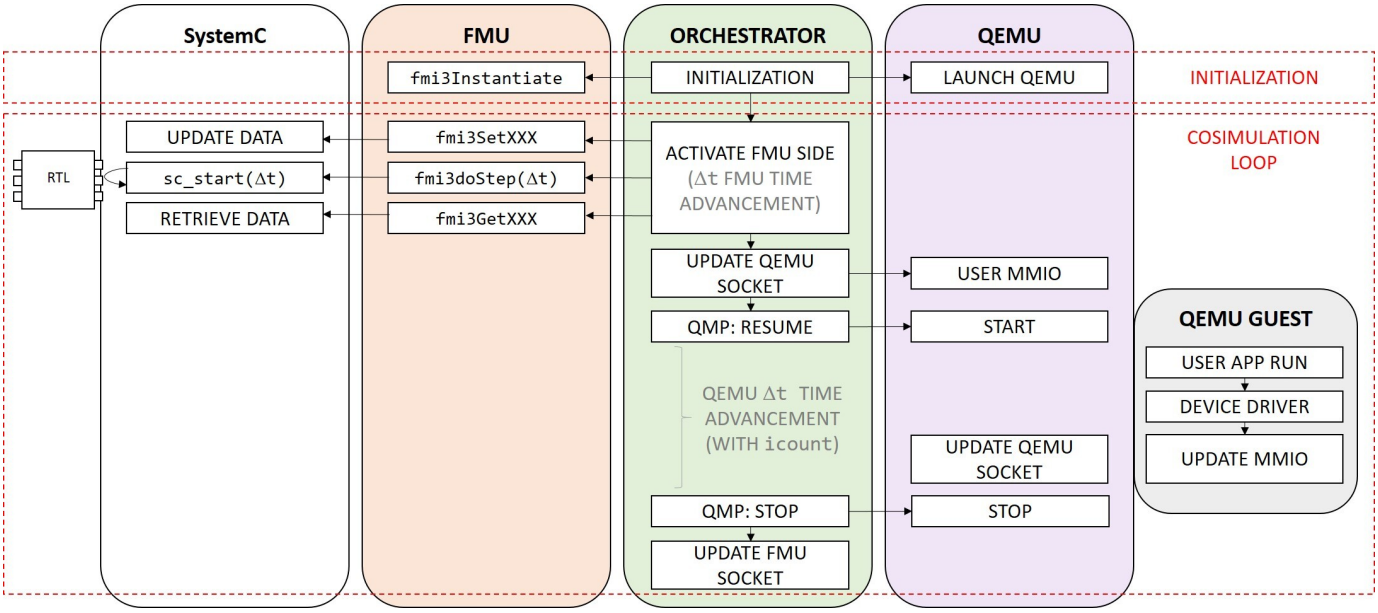


Fig. 3. **Overview of the co-simulation framework. The orchestrator instantiates both domains (QEMU and FMU), then executes a time-synchronized loop: FMU receives updated input and runs for Δt; the results are copied to the QEMU MMIO area; QEMU runs for Δt executing the user application, eventually updating the MMIO through the device driver.**

transfer data values from and to the MMIO memory region, and one for QMP control commands, used to activate and suspend QEMU execution. FMU-side communication requires one socket, used for transferring data from and to the FMU, and to activate the FMU API. The master handles the co-simulation algorithm, by initializing both QEMU and the FMU and by periodically advancing their respective simulated time.

2) Orchestration model: The orchestration layer defines a global execution model that aligns QEMU instruction-driven behavior with FMI step-based progression. This model complements the abstraction-specific semantics of `fmi3DoStep` defined in Section IV-B2a, lifting them to a system-level synchronization mechanism and providing a consistent global notion of time across domains. The co-simulation follows a *lock-step execution scheme*, where each iteration corresponds to a single FMI communication step Δt. During each step, the orchestration layer ensures that both QEMU and the FMU advance by an equivalent amount of simulated time. By enforcing synchronization at each communication step, the orchestration ensures that all domain-specific events are resolved before advancing time, preserving causality and consistency.

a) Initialization phase: The orchestrator initializes the system by launching QEMU and establishing external control via QMP, while concurrently instantiating and initializing the FMU through the FMI API. This ensures that both execution domains start from a consistent state before entering the synchronized co-simulation loop.

b) Co-simulation loop: Each co-simulation iteration implements a request–response synchronization cycle:

- FMU simulation for time Δt is advanced by invoking `fmi3DoStep`, whose internal behavior depends on the SystemC abstraction level (Section IV-B2a);
- the generated output is mapped to the FMU socket, and is used by the orchestrator to feed the QEMU data socket;
- the updated data are transferred to the MMIO region;
- QEMU execution is resumed for a bounded interval, corresponding to the communication step Δt;
- the application resumes execution; whenever needed, the application interacts with the virtual devices through MMIO operations via the device driver;
- any operation on the MMIO is converted into data communications via the QEMU data socket;
- when the fixed time step expires, QEMU execution is suspended, and the master propagates the new data onto the FMU socket, to close the loop.

This loop establishes a synchronized request–response interaction, ensuring that all observable effects within a communication step are resolved before proceeding.

E. Determinism and scheduling control

QEMU is operated in `icount` mode to enforce deterministic execution and to enable predictable advancement of simulated time. External control through QMP allows the orchestration layer to introduce explicit synchronization points, resolving the scheduling mismatch between QEMU's internal execution loop and the FMI co-simulation model. This guarantees reproducibility across runs and ensures consistent alignment with the `fmi3DoStep` semantics.

Through this coordinated execution model, the orchestration layer provides a unified notion of time across heterogeneous domains, causality preservation between software and hardware interactions, and deterministic and reproducible co-simulation behavior.

F. Interrupt propagation from SystemC to FMI master

Interrupts are a critical aspect, as they may occur inside a Δt time window and thus they may not be intercepted

TABLE III
CHARACTERISTICS OF SYSTEMC MODELS USED FOR FMI WRAPPING.

Design	Level	#LOC	FMI Interface		ISS Integration	
			#Ports/Vars	Supported Mechanisms	#LOC	#doStep
Register-Transfer Level (RTL) Models						
ADPCM	RTL	922	4 IN / 4 OUT	clock, `sc_stop()`	988	Table V
CRC	RTL	2,467	10 IN / 3 OUT	clock	770	73
Transaction-Level (TLM) Models						
ECC	TLM	1,311	7 PAYLOAD FIELDS	-	754	103
eMMC	TLM	6,078	22 PAYLOAD FIELDS	`wait()`	806	6
I2C	TLM	1,072	7 PAYLOAD FIELDS	interrupt, mixed `b_/nb_transport()`	744	2500
Mixed-Level Models						
GPIO controller	RTL + TLM	697	10 IN /3 OUT	interrupt, clock, `wait()`	883	4034
Analog/Mixed-Signal (AMS) Models						
Delta Sigma + Voltage Source	RTL + AMS	522	7 IN (4 AMS, 3 RTL) / 3 OUT (1AMS, 2 RTL)	-	160	2000
Harvester	AMS	1,100	1 SCA_IN / 1 SCA_OUT	ELN, TDF, LSF mix	711	25000

and handled by the ISS in a timely manner. To avoid this, the orchestrator starts computation from the FMU side. In the initial configuration file, one signal may be labeled as potential interrupt, and is treated with special care during FMU construction. If a `write` operation is applied to it, the FMU does not reach the end of the Δt, but rather enters `EventMode` and returns the reached time stamp $\Delta t_1 < \Delta t$. This time is communicated to the orchestrator, that activates QEMU so to reproduce an advancement of Δt_1. In this way, the interrupt can be timely handled by the interrupt service routine at the next co-simulation loop. From that moment on, co-simulation goes back to the predefined time step Δt, and resumes normal execution.

G. Integration of various execution models

As stated in Section II-D, integrating (different flavours of) SystemC, QEMU, and FMI introduces fundamental mismatches. Given that FMI is used as a bridge between SystemC and QEMU, the proposed solution enforces the FMI step-based progression as co-simulation synchronization and semantics. This implies that, as in the FMI standard, an external master sequentially activates each actor so to advance its computation and local time by Δt. This is translated into different solutions, specific of the semantics of each actor. For QEMU, this translates into using `icount` to make a temporal estimation of software execution, and in using the QMP protocol to activate and pause ISS execution (Section IV-C). The activation of SystemC is realized through an invocation of `sc_start`(Δt), with different checks and implementation strategies specific of the abstraction level. However, all SystemC-based execution models (including TLM, RTL and AMS) are natively handled by the SystemC scheduler, that maps all behaviors to an event- and time-based synchronization mechanism.

V. EXPERIMENTAL VALIDATION

We evaluated the proposed framework across eight case studies spanning RTL, TLM, and AMS abstraction levels, focusing on semantic preservation and co-simulation performance. All case studies have been provided by an industrial partner, ensuring representative design complexity and practical relevance, and they have been selected to support a large variety of abstraction levels and supported mechanisms. Table III characterizes the eight designs used for validation in terms of level of abstraction, lines of code (LOC), interface, specific mechanisms (e.g., interrupt, clock), and number of invocations of the `fmi3DoStep` function.

All experiments were executed on a Windows 11 host equipped with an Intel Core i7-12700H processor and 16 GB of RAM. QEMU-based emulation ran inside Windows Subsystem for Linux 2 (WSL2) on Ubuntu 24.04.2 LTS, configured with 4 virtual CPUs and 4GB guest memory. Both ARM and RISC-V targets were supported. Code generations took few seconds for each design.

The primary objective of the experimental evaluation is to verify that the proposed framework *preserves the functional and timing semantics of the original SystemC models* across all execution environments. To this end, each design is evaluated in three configurations: native SystemC simulation, FMI-based co-simulation, and vHIL execution with embedded software running on a QEMU-based platform. Across all evaluated designs execution results proved to be *bit-accurate and to respect simulated time accuracy* with respect to the native SystemC reference.

Execution time (Table IV) and memory footprint are reported as characterization metrics, not optimization targets. For time execution, total time includes QEMU boot, in-guest driver and application deployment, FMU loading, and full simulation, while simulation time isolates the co-simulation loop. Two overhead ratios are used: FMU wrapping cost $X = \frac{T_{\text{FMU}}}{T_{\text{SystemC}}}$, and full-system emulation cost $Y = \frac{T_{\text{vHIL}}}{T_{\text{FMU}}}$.

A. ADPCM Encoder

The Adaptive Differential Pulse-Code Modulation (ADPCM) is an industrial encoder that accepts 16-bit signed PCM samples and produces 4-bit encoded values. The SystemC RTL design contains a synchronous `SC_METHOD` clocked at $1\,\text{ns}$, decomposed into eleven states that encode each sample in a fixed pipeline. The ADPCM is evaluated over twelve configurations spanning $P \in \{1, 2, 5, 25\}$ packets and $S \in \{200, 500, 1000\}$ samples each.

Supporting the ADPCM required to use the clock support mechanism in Section IV-B4b. The industry-provided testbench invokes `sc_stop()` in case of severe errors, which

TABLE IV
EXECUTION TIME RESULTS (S) ACROSS ALL CASE STUDIES. *sim* = SIMULATION LOOP TIME (S); *tot* = TOTAL WALL-CLOCK TIME (S), INCLUDING QEMU BOOT (140 TO 194 S) AND DRIVER COMPILATION (43 S).

Design	Level	SystemC (s)		FMU (s)			vHIL RISC-V (s)			vHIL ARM (s)		
		sim	tot	sim	tot	X	sim	tot	Y_{RV}	sim	tot	Y_{ARM}
Register-Transfer Level (RTL) Models												
ADPCM (min, $P1, S200$)	RTL	0.23	0.23	1.23	1.25	5.35×	10.8	243	8.8×	10.2	241	8.3×
ADPCM (max, $P25, S1,000$)	RTL	27.0	27.0	93.5	93.5	3.6×	1,974	2,210	21.1×	1,860	2,096	19.9×
CRC	RTL	0.004	0.004	0.025	0.048	6.25×	1.36	280	54.3×	1.28	276	51.3×
Transaction-Level (TLM) Models												
ECC	TLM	0.011	0.011	0.070	0.136	6.4×	0.70	109	10.1×	2.31	237	33.2×
eMMC	TLM	2.22	2.22	3.75	3.75	1.69×	9.01	247	2.40×	8.48	247	2.26×
I2C	TLM	0.63	0.63	1.16	1.18	1.85×	3.92	114	3.38×	5.8	247	5×
Mixed-Level Models												
GPIO controller	RTL + TLM	0.0012	0.002	0.034	0.055	28.3×	1.088	120	32×	1.012	234	29.78×
Analog/Mixed-Signal (AMS) Models												
Delta Sigma + Voltage Source	RTL + AMS	1.46	1.52	3.019	3.111	2.07×	9.86	297	3.27×	6.71	201	2.22×
Harvester	AMS	1.85	1.85	6.43	6.46	3.48×	52.59	247	8.18×	51.96	191	8.08×

required the halt mechanism in Section IV-B4a. Both conditions were annotated in the initial configuration file, and automatically supported.

TABLE V
ADPCM ENCODER: EXECUTION TIME (S) AND OVERHEAD RATIOS.

Config	SystemC		FMU		X	vHIL (RISC-V)		vHIL (ARM)		Y_{riscv}	Y_{arm}
	Sim.	Tot.	Sim.	Tot.		Sim.	Tot.	Sim.	Tot.		
$P1, S200$	0.23	0.23	1.23	1.25	5.35	10.8	243	10.2	241	8.8	8.3
$P1, S500$	1.00	1.00	1.50	1.53	1.53	19.5	276	18.4	274	13.0	12.3
$P1, S1000$	1.27	1.27	3.05	3.08	2.43	34.1	272	32.1	270	11.2	10.5
$P2, S200$	0.48	0.48	1.14	1.17	2.42	13.1	245	12.4	243	11.5	10.9
$P2, S500$	1.04	1.04	2.86	2.89	2.78	34.5	262	32.6	260	12.1	11.4
$P2, S1000$	2.14	2.14	5.45	5.50	2.57	83.8	330	79.1	326	15.4	14.5
$P5, S200$	1.04	1.04	2.81	2.86	2.75	38.2	271	36.0	269	13.6	12.8
$P5, S500$	2.46	2.46	7.69	7.74	3.15	109.0	371	102.8	365	14.2	13.4
$P5, S1000$	5.13	5.13	15.4	15.5	3	244.8	490	230.8	476	15.9	15.0
$P25, S200$	5.0	5.0	15.7	15.7	2.93	259.9	598	244.8	583	16.6	15.6
$P25, S500$	12.0	12.0	43.3	43.3	3.62	820.6	1056	772.9	1009	19.0	17.9
$P25, S1000$	27.0	27.0	93.5	93.5	3.40	1974.3	2210	1859.7	2096	21.1	19.9

Detailed execution time statistics are reported in Table V. The FMU wrapping overhead X ranges from $1.5\times$ to $5.35\times$, while the vHIL slowdown Y grows with workload size, reflecting the latency of one co-simulation loop (including the management of MMIO, interrupts, and socket communication). The ARM target is consistently 5–7% faster in simulation owing to superior QEMU TCG translation efficiency (Y is from $8.8\times$ to $21.1\times$ for RISC-V, and from $8.3\times$ to $19.9\times$ for ARM). Total vHIL execution is dominated by guest boot (194 s) and in-guest driver compilation (43 s), which are one-time fixed costs independent of workload and nearly identical across both targets. Memory footprint is stable across all configurations: the FMU layer adds $73\times$ VM overhead relative to native SystemC (952–955 MB vs. 13 MB), while RSS remains modest at 42-50 MB; both vHIL targets reach 2 GB RSS, dominated by the QEMU guest image, with ARM marginally lower ($8.2\times$ vs. $8.7\times$ relative to FMU RSS) due to a leaner kernel image.

B. CRC

The CRC is an RTL implementation of a multi-context cyclic redundancy check (CRC) engine supporting up to 16 independent computation contexts, with input width of either 8, 16 or 32 bits. Correct operation requires a minimum latency of one clock cycle between a write to `CRC_INP` and a read of `CRC_OUTP`; the FMI master preserves this by mapping each clock to a `fmi3DoStep` call of one clock period. The CRC is clocked, and thus its support required the clock support mechanism in Section IV-B4b.

The test suite includes six test sequences covering CRC-16 and CRC-32 operation. FMU simulation incurs a $6.25\times$ slowdown over native SystemC. vHIL execution is dominated by full-system emulation: both RISC-V and ARM are 2.5 orders of magnitude slower than the SystemC baseline. Memory overhead is similar as for the ADPCM case study: the FMU requires $67.6\times$ more VM and $7.55\times$ more RSS than native SystemC, while VHIL RSS grows by a further $53\times$.

C. Error Correction Code Engine

The ECC is a hardware-accurate Hamming-based Error Correction Code (ECC) engine for NAND flash memory, modeled at SystemC TLM. The module exposes a single target socket implementing the blocking transport protocol, wrapping 7 fields for control and data.

The ECC engine was validated using a total of 103 unique 16-bit input vectors; each vector was injected through the `b_transport()` interface, triggering a complete ECC computation within a single TLM transaction. The resulting 32-bit codeword was compared against a reference software model implementing the identical Hamming SECDED scheme. All 103 test cases passed, confirming bit-accurate agreement. Given that the design is totally untimed, both the RISC-V and ARM were configured without `icount` QEMU mode, reducing emulator startup latency and relying on blocking calls for synchronization.

Wrapping the TLM logic as an FMU introduces a factor of approximately $6.4\times$ in total execution time, mainly due to FMI 3.0 marshalling and shared-library loading. The total execution time ratio ($6.4\times$) exceeds both the init and simulation ratios, indicating additional overhead from FMI lifecycle setup and teardown phases outside the pure simulation loop.

TABLE VI
eMMC CONTROLLER: PER-TRANSACTION TYPE OVERHEAD RATIOS

Transaction	**JEDEC commands**	X	Y_{riscv}	Y_{arm}
`initiate`	CMD0-CMD3	3.18	14.6	13.7
`setup_data_transfer`	CMD 7, CMD16	2.64	20.3	19.2
`write_data`	CMD24	2.47	2.05	1.93
`read_data`	CMD17	1.23	2.74	2.58
`erase_data`	CMD35	1.85	3.19	3.01
`trim_data`	CMD38	0.66	2.53	2.39

The vHIL platform incurs a further $10.1\times$ slowdown over the FMU baseline, driven by full ISA emulation, operating system execution, and software-driven polling of the memory-mapped interface. From a memory perspective, the FMU requires approximately $78\times$ more VM, while RSS increase is more moderate ($15\times$).

D. eMMC Controller

The eMMC controller models the command, data, and control paths of a JEDEC-compliant embedded MultiMediaCard device at TLM. Supporting the eMMC required to support the `wait()` mechanism, as in Section IV-B4c.

Functional validation uses six host-driven transaction sequences exercising the full command and data path (Table VI). FMU wrapping overhead X varies across workloads: control-dominated phases (e.g., `initiate` and `setup_data_transfer`) exhibit higher relative overhead due to short simulation intervals amplifying the fixed per-step synchronisation cost, while data-intensive transactions amortize this cost over longer execution.

vHIL slowdown Y peaks at $20.3\times$ for `setup_data_transfer`, where frequent command interactions dominate, and drops to $2.05\times$ for `write_data`. Memory overhead follows trends consistent with previous case studies: FMU VM grows by $75.6\times$ relative to native SystemC, with RSS increasing by $13.5\times$. vHIL RSS reaches approximately 1.2GB for RISC-V and 1.1GB for ARM, dominated by the QEMU guest environment.

E. I2C-Interfaced ALU

This case study evaluates a TLM-2.0 model of an I2C-interfaced arithmetic logic unit (ALU) and a Register File. The architecture comprises three loosely-coupled modules using a mix of blocking and non-blocking transport primitives. This design requires support for the interrupt mechanism, that notifies the end of data transfers as in Section IV-F.

Functional validation was conducted over 2,500 randomized transactions covering all ALU opcodes and boundary operand values. The register state after each transaction matched a reference software model with bit-exact accuracy, confirming correctness of both the datapath and the non-blocking communication protocol across all execution environments.

The FMU simulation overhead is $1.84\times$ for simulation and $1.85\times$ overall. This represents a moderate performance degradation relative to native SystemC execution despite the transaction-oriented nature of the design. Across 2,500 randomized transactions, the average FMU step execution time is 0.462 ms. The additional runtime is attributable to the co-simulation infrastructure required by the FMI-based execution flow, including communication and synchronization overheads between the FMU and the simulation environment. However, the overhead remains limited, indicating that the proposed methodology scales effectively to TLM-2.0 designs combining blocking and non-blocking transport mechanisms.

The vHIL simulation overhead is $3.38\times$ for RISC-V and $5.0\times$ for ARM. While the virtual hardware layer introduces a significant performance penalty relative to native SystemC and FMU execution, the resulting execution times remain suitable for transaction-level functional verification. As in the other case studies, total execution time includes a fixed cost of approximately 237 s associated with QEMU boot and driver compilation. The difference between the RISC-V and ARM results indicates that the vHIL overhead is influenced not only by the communication workload of the TLM model but also by architecture-specific characteristics of the virtual platform and software environment.

Memory overhead follows the pattern established by prior TLM designs: FMU grows VM by $81.7\times$ and RSS by $28.5\times$. At the vHIL level, RSS remains notably modest ($1.14\times$ for RISC-V and $0.69\times$ for ARM), indicating that the guest OS and QEMU device model add minimal resident memory pressure beyond the FMU process for this design.

F. Mixed-Signal GPIO Controller

This case study evaluates a *mixed-signal GPIO controller* that integrates two abstraction levels within a single SystemC module: a TLM-based SPI register interface and an RTL datapath. The design reflects common industrial practice, where transaction-level and signal-level components coexist, share internal state, and are tightly coupled through causal interactions. The SystemC implementation consists of concurrent processes triggered by events such as reset, SPI transactions, and interrupts. These processes directly connect TLM activity to signal-level behavior, ensuring that register writes immediately affect signal routing, while signal transitions propagate within the same delta cycle.

FMU wrapping preserves this execution semantics without modification. Delta cycles, event scheduling, and process activation remain unchanged, with only the external interface adapted to the FMI standard. Consequently, the full causal chain (from transaction-level operations to signal-level transitions) is executed within a single `fmi3DoStep`, maintaining behavioral equivalence with native SystemC simulation. Delayed events (e.g., interrupts and `wait()` statements) are also handled correctly and consistent.

The FMU simulation exhibits a $28.3\times$ slowdown compared to native SystemC (0.034s vs. 0.0012s), primarily due to per-step synchronization overhead. This effect is amplified in scenarios with frequent fine-grained interactions, where a large number of small `fmi3DoStep` calls are required.

vHIL overhead follows the same pattern observed in previous case studies. Simulation time is $32\times$ slower on RISC-V and $29.8\times$ slower on ARM relative to the FMU. Total execution time is dominated by platform overheads, reaching

120s (RISC-V) and 234s (ARM), largely due to QEMU boot time and driver compilation.

Memory overhead remains consistent with earlier observations: the FMU requires significantly more virtual memory than native SystemC, while vHIL resident memory usage reaches approximately 1.2GB on both targets.

G. Delta-Sigma Modulator and Voltage Source

The Delta-Sigma experiment couples a configurable sine-wave AMS voltage source with a first-order $\Delta\Sigma$ modulator (DSM) described at the RTL level, that evaluates the instantaneous output at each TDF timestep ($\Delta t = 1\,\mu\text{s}$). The resulting analog waveform is then fed into the DSM, which produces a 1-bit bitstream suitable for downstream digital processing. Each subsystem is packaged as an independent FMU and evaluated in two configurations: a *standalone* (FMU-only) execution, and a *virtual vHIL setup* where a guest application running inside QEMU runs the software part of the system.

Both configurations were evaluated with $f = 500\,\text{Hz}$, $A = 0.9\,\text{V}$, $\varphi = 0$, and $h = 1\,\mu\text{s}$, for a total simulated duration of $2\,\text{ms}$ (2,000 steps, corresponding to one sine period).

The FMU-only configuration completes $2\,\text{ms}$ of simulated time in $3.111\,\text{s}$ wall-clock time. In contrast, the vHIL configurations introduce significant overhead due to cross-domain synchronization, with $3.27\times$ at simulation time for RISC-V and $2.2\times$ for ARM. This improvement is reflected in the reduced simulation time ($6.707\,\text{s}$ vs. $9.862\,\text{s}$) and total execution time ($201.449\,\text{s}$ vs. $297.2\,\text{s}$). A substantial fraction of the total vHIL execution time is dominated by guest initialization.

H. Electrostatic Energy Harvester

As a last experiment, the Harvester design is a SystemC-AMS model of an electrostatic vibration energy harvester, integrating TDF, ELN and LSF constructs within a unified hierarchical description, with a TDF interface. The model is validated using a workload of 25,000 simulation steps with a global timestep of $4\,\text{ns}$, corresponding to a total simulated duration of $100\,\mu\text{s}$.

The FMU simulation overhead $X = 3.48\times$ is the lowest among all digital and mixed-signal case studies, confirming the expected behavior: for computationally intensive continuous-time models, numerical integration dominates over co-simulation synchronization overhead. As a result, the cost of each co-simulation loop is effectively amortized across the 25,000 `fmi3DoStep` invocations. This contrasts with command-intensive TLM designs such as eMMC, where short simulation intervals amplify fixed synchronization costs.

The vHIL slowdown $Y = 8.1\times$ on both targets is consistent with the Delta-Sigma Modulator results, and confirms that for AMS workloads, full-system emulation overhead is primarily determined by the per-step communication pipeline rather than ISA-specific translation efficiency. The near-identical RISC-V and ARM vHIL ratios indicate that the dominant cost is continuous-time numerical computation within the FMU, rather than guest instruction throughput. From a memory perspective, the FMU requires approximately $30\times$ more VM, while vHIL RSS reaches approximately $870\,\text{MB}$ on both targets (an $18\times$ increase over the FMU baseline).

VI. Discussion and Key Findings

The experimental evaluation highlights several key insights on the behavior of the proposed framework.

A. Semantic Preservation and Automatic Integration

Across all evaluated designs, spanning RTL, TLM, and AMS models, the framework consistently preserves both functional correctness and abstraction-specific execution semantics: clocked RTL designs maintain cycle-accurate behavior, TLM models preserve transaction ordering and latency, and AMS models retain continuous-time dynamics. The case studies also exercise a broad set of mechanisms, including interrupts, clocks, and mixed ELN/LSF primitives. Successful generation across all designs shows that the framework is not tied to a specific abstraction level or modeling style, and applies to heterogeneous SystemC descriptions with minimal user intervention.

A key characteristic of the proposed solution is its support for the development workflow: developers need only provide the SystemC description and a configuration file, while all integration artifacts (FMU wrapper, `modelDescription.xml`, virtual device, software driver, orchestration infrastructure, and application skeleton) are generated automatically. This substantially reduces the manual effort traditionally required to construct HW/SW virtual platforms, lowers the risk of integration errors, and lets developers focus on model and application development rather than platform integration.

B. Performance

The dominant factor influencing FMU execution overhead is communication step granularity Δt. Fine-grained workloads, such as interrupt-driven or event-dense designs, incur higher overhead due to frequent synchronization and communication, while computation-intensive models amortize this cost over longer execution intervals. This behavior is consistent across all abstraction levels and highlights the fundamental trade-off between temporal resolution and simulation efficiency.

The transition from FMU co-simulation to vHIL execution introduces additional overhead, primarily driven by QEMU-based full-system emulation, including operating system execution, device interaction, and communication latency. The co-simulation loop becomes the dominant cost in command-intensive scenarios, whereas its impact is reduced in computation-heavy workloads. One-time costs such as QEMU boot and software deployment significantly influence total execution time but are amortized over longer simulations. Behavior is consistent across both RISC-V and ARM targets (minor differences from QEMU translation efficiency) and stable across all designs, confirming that the framework scales across modeling paradigms, interaction patterns, and computational characteristics.

The observed overheads are not introduced by the framework itself, but are largely inherent to virtual-platform construction and cross-domain co-simulation: any conventional workflow would incur the costs of building the QEMU environment, device models, kernel drivers, and synchronization logic. The framework does not eliminate these costs,

but addresses them through systematic automation within a correct-by-construction integration flow that allows developers to extend and validate application software on a reliable and reproducible virtual platform.

C. Anatomy of a co-simulation cycle

To further analyse the induced overhead, Fig. 4 breaks down the per-cycle latency budget for the I2C (Section V-E), a transaction-heavy TLM design. The mean inter-request interval of 1.845 ms splits almost evenly between `fmi3DoStep` (46 %) and the QEMU guest-side pipeline (47 %), the latter covering the TCP round-trip, interrupt handling, and the user-space `read()` wake-up. FMI marshalling is marginal: the `fmi3Set` calls account for 3 % and the `fmi3Get` output reads for 2 %, while TCP transmission and Python orchestration together add the remaining 2 %. This decomposition reveals a fundamental property of the proposed architecture: co-simulation overhead is not concentrated in the integration layer. Instead, the cycle budget is dominated by two costs that are inherent to any equivalent virtual platform: model computation (`fmi3DoStep`) and full-system ISA emulation (QEMU guest pipeline), each contributing roughly half of the total latency. FMI communication and orchestration layers together account for less than 10 % of the cycle, confirming that the framework introduces limited integration overhead.

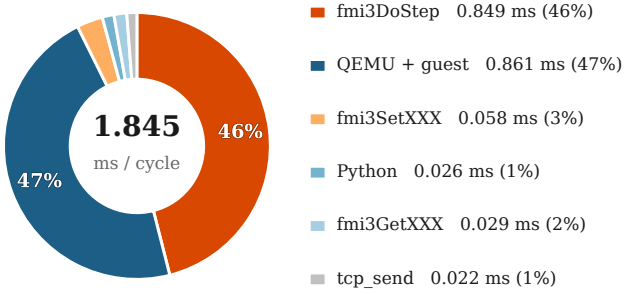


Fig. 4. Single-cycle time breakdown of the I2C vHIL co-simulation on RISC-V, considering mean latencies over 2,500 steps.

To separate the cost of FMI specifically, we evaluated a direct-coupling configuration that retains the co-simulation infrastructure but bypasses FMI variable marshalling and the `fmi3DoStep` dispatch chain. In Fig. 5, the gap between the QEMU + SystemC bars and the vHIL bars decomposes into two separable sources. The first is intrinsic to the co-simulation architecture: synchronising SystemC and QEMU through a lock-step request-response loop adds latency regardless of the integration standard, and already pushes the QEMU + SystemC baseline above native SystemC. The second is the FMI layer itself (variable marshalling, value-reference resolution, and the `fmi3DoStep` dispatch) which adds a bounded cost on top and accounts for the residual gap.

Removing FMI to improve performance would eliminate the framework's defining capability: composing models from potentially heterogeneous tools and abstraction levels under a single standardized master. The measured gap is thus the price of interoperability, rather than implementation inefficiency.

Performance overhead could nonetheless be reduced by reducing the marshalling and communication load, e.g., by packing multiple SystemC ports into a single FMI variable (e.g., via the `__attribute__((__packed__))` construct). This will be investigated as future work.

D. Step-size relaxation

Lock-step synchronisation is the framework's primary bottleneck. The minimum step Δt for each case study is set by the temporal granularity of the target device and the application: a device that consumes inputs and produces outputs every clock cycle (e.g. the CRC engine) forces $\Delta t \leq T_{\text{clk}}$ and a correspondingly frequent loop, whereas a device whose datapath spans several cycles before producing an observable output admits a coarser Δt, reducing `fmi3DoStep` invocations and communication proportionally.

The ADPCM allows to prove this idea, as its RTL state machine takes exactly 7 cycles per sample, with no input consumed or output observable in between, so $\Delta t = 7\,T_{CLK}$ is semantically equivalent while cutting TCP round-trips and FMI calls sevenfold. Table VII reports the resulting speed-up at S=200: 3.5× at P=5, and 4.4× at P=10 and P=25. The speed-up saturates near 4.4× once the workload amortizes the fixed communication cost.

TABLE VII
ADPCM: SIMULATION TIME SPEED-UP FROM STEP-SIZE RELAXATION ($\Delta t = 7\,T_{\text{clk}}$, RISC-V, $S = 200$).

Config	Baseline (s)	Relaxed (s)	Speed-up
$P5$, $S200$	38.2	10.8	3.5×
$P10$, $S200$	84.9	19.3	4.4×
$P25$, $S200$	259.9	58.9	4.4×

The support for interrupt management provided by the framework (Section IV-F) ensures that correctness is preserved under asynchronous events when exploiting large Δt values. The interrupt mechanism allows indeed the FMU to exit a co-simulation step early returning the actual elapsed time $\Delta t_1 < \Delta t$ at which the event occurred, and to immediately propagate control to the software stack for interrupt service. This ensures that asynchronous notifications are never delayed beyond their natural occurrence time, regardless of the configured step size.

To conclude, co-simulation performance is a function of the input configuration: the choice of Δt, the presence of interrupts, and the temporal structure of the target device collectively determine the trade-off between simulation speed and temporal resolution. The proposed framework exposes these degrees of freedom through the configuration file, allowing the user to tune synchronisation granularity without modifying the SystemC model, the generated wrapper, or the software stack.

E. FMI as Integration Backbone

The communication and execution overheads introduced by FMI are compensated by the advantages of a standardized integration backbone: (i) it enables interoperability with any FMI-compliant environment without requiring custom adapters; (ii) this naturally scales beyond hardware/software co-simulation,

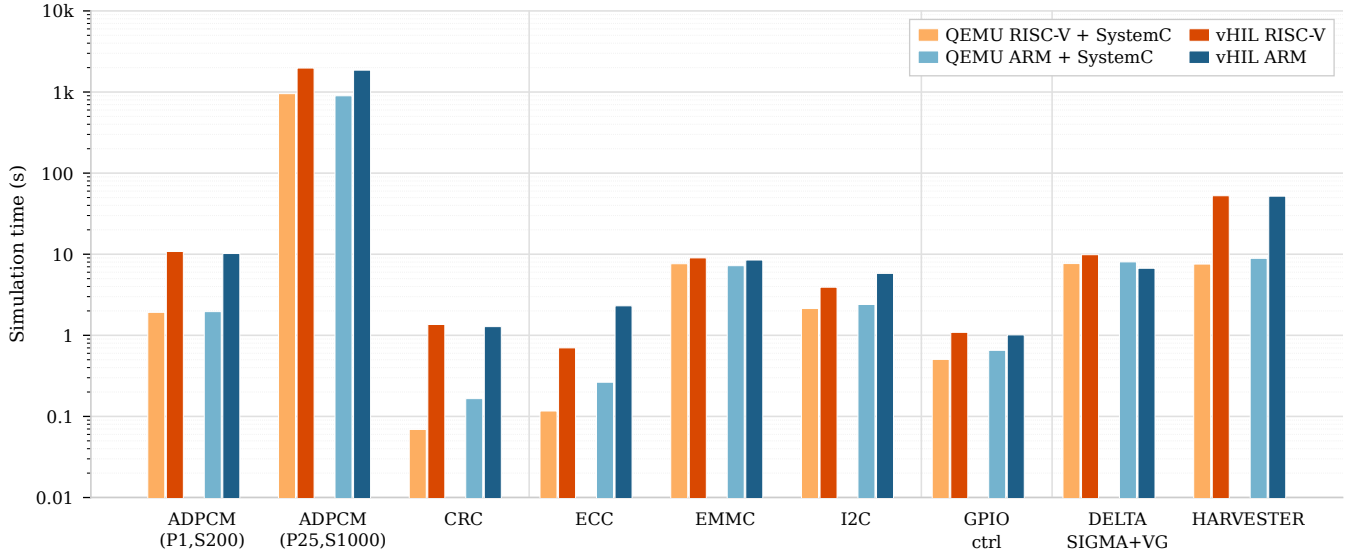


Fig. 5. Comparison of QEMU + native SystemC co-simulation against the full vHIL framework on both RISC-V and ARM targets.

allowing integration of additional independent FMUs without modifying the underlying orchestration mechanism. (iii) FMI also enforces a clear separation between model semantics and platform integration, where the orchestration layer remains independent of both hardware and software implementations. This transforms a point-to-point HW/SW co-simulation setup into a reusable multi-domain virtual platform architecture.

F. Overall Assessment

To conclude, the framework is not only an automatic integration flow, but it rather provides a unified co-simulation methodology that reconciles heterogeneous execution models (SystemC, QEMU, and FMI) while preserving their native semantics. This combination of automation and semantic correctness, plus the FMI-based extensible backbone, distinguishes the proposed approach from existing ad-hoc integration solutions. Overall, the results demonstrate that the proposed framework provides a robust and predictable foundation for cross-domain co-simulation, enabling accurate integration of heterogeneous SystemC models with ISS software execution.

VII. Conclusion

This paper presented an automated, open-source framework for FMI 3.0-compliant virtual platforms integrating heterogeneous SystemC models with QEMU-based ISS simulation. Eight case studies across all abstraction levels demonstrated bit-accurate and temporally consistent behaviour across native, FMI, and vHIL environments. Results show that FMU overhead is driven by co-simulation granularity, while vHIL overhead is dominated by communication latency, with lower impact for computation-intensive workloads. Future work will focus on parallelization of FMU execution, on introducing adaptive step-size control, and on extending support to additional processor architectures, to broaden applicability in automotive and cyber-physical system design workflows. Additionally, the FMI backbone will allow the generated virtual platforms to be extended with additional FMUs, without modifying the orchestration mechanism.

Acknowledgment

The authors used an AI-assisted writing tool (Claude) to refine the manuscript and improve readability. All technical content, analysis, and conclusions were developed and verified by the authors.

References

[1] S. A. Seshia, S. Hu, W. Li, and Q. Zhu, “Design automation of cyber-physical systems: Challenges, advances, and opportunities,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1421–1434, 2017.

[2] S. Sonko, C. Daudu, F. Osasona, A. Monebi Ph.D., E. Etukudoh, and A. Atadoga, “The evolution of embedded systems in automotive industry: A global review,” *World Journal of Advanced Research and Reviews*, vol. 21, pp. 96–104, 02 2024.

[3] T. S. Madhuri and J. Bala Vishnu, “Software-defined vehicles: The future of automobile industry,” in *Proc. of COMSNETS*, 2025, pp. 192–197.

[4] A. Löcklin, M. Müller, T. Jung, N. Jazdi, D. White, and M. Weyrich, “Digital twin for verification and validation of industrial automation systems – a survey,” in *Proc. of IEEE ETFA*, vol. 1, 2020, pp. 851–858.

[5] A. Rajhans, A. Bhave, I. Ruchkin, B. H. Krogh, D. Garlan, A. Platzer, and B. Schmerl, “Supporting heterogeneity in cyber-physical systems architectures,” *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3178–3193, 2014.

[6] KPIT Technologies, “Accelerating vehicle development: Overcoming verification and validation hurdles through virtual validation & simulation,” https://www.kpit.com, 2025.

[7] Modelica Association, “Functional Mock-up Interface (FMI).” [Online]. Available: https://fmi-standard.org/

[8] Z. Zhou, X. Zhou, L. Du, P. Ma, X. Wang, Y. Chen, M. Liu, T. Wang, L. Hui, and C. Zeng, “Simulation of embodied cyber physical system based on Modelica/MWORKS: A case study of intelligent unmanned surface vessel,” in *Proc. of Modelica Conference*, 10 2025.

[9] E. Widl, W. Müller, D. Basciotti, S. Henein, S. Hauer, and K. Eder, “Simulation of multi-domain energy systems based on the functional mock-up interface specification,” in *Proc. of EDST*, 2015, pp. 510–515.

[10] M. Wiens, T. Meyer, and P. Thomas, “The potential of FMI for the development of digital twins for large modular multi-domain systems,” in *Proc. of Modelica Conference*, 09 2021, pp. 235–240.

[11] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: A survey,” *ACM Comput. Surv.*, vol. 51, no. 3, May 2018.

[12] System C Standardization Working Group, “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pp. 1–618, 2023.

[13] SystemC AMS extensions Working Group, "IEEE Standard for Standard SystemC Analog/Mixed-Signal Extensions Language Reference Manual," *IEEE Std 1666.1-2016*, pp. 1–236, 2016.

[14] E. Fraccaroli and S. Vinco, "Modeling Cyber-Physical Production Systems With SystemC-AMS," *IEEE Transactions on Computers*, vol. 72, no. 07, 2023.

[15] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proc. of USENIX*, USA, 2005, p. 41.

[16] M. H. Quraishi, H. S. Sarjoughian, and S. Gholami, "Co-simulation of hardware RTL and software system using FMI," in *Proc. of WSC*, 2018, pp. 572–583.

[17] S. Centomo, J. Deantoni, and R. De Simone, "Using SystemC cyber models in an FMI co-simulation environment: Results and proposed FMI enhancements," in *Proc. of Euromicro DSD*, 2016, pp. 318–325.

[18] S. Centomo, M. Lora, and F. Fummi, "Transaction-level functional mockup units for cyber-physical virtual platforms," in *Proc. of FDL*, 2018, pp. 5–8.

[19] R. L. Bucs, L. G. Murillo, E. Korotcenko, G. Dugge, R. Leupers, G. Ascheid, A. Ropers, M. Wedler, and A. Hoffmann, "Virtual hardware-in-the-loop co-simulation for multi-domain automotive systems via the functional mock-up interface," in *Proc. of FDL*, 2015, pp. 1–8.

[20] S. E. Saidi, A. Charif, T. Sassolas, P.-G. Le Guay, H. V. Souza, and N. Ventroux, "Fast virtual prototyping of cyber-physical systems using SystemC and FMI: ADAS use case," in *Proc. of RSP*. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–49.

[21] M. Safar, K. Khalil, M. A. El-Moursy, and M. Abdelsalam, "Virtual electronic control unit as a functional mockup unit for heterogeneous systems," in *Proc. of IEEE ISCAS*, 2018, pp. 1–5.

[22] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast virtual prototyping for embedded computing systems design and exploration," in *Proc. of RAPIDO*. New York, NY, USA: Association for Computing Machinery, 2019.

[23] J. H. Weinstock, L. Jünger, S. Winther, N. Bosbach, N. Zurstraßen, A. Salios, R. Brandhofer, P. R. Andrade, and rpelke, "Virtual components modeling library (vcml)," https://github.com/machineware-gmbh/vcml, mar 13 2026.

[24] G. Pollo, A. M. Albu, A. Burrello, D. J. Pagliari, C. Tesconi, L. Panaro, D. Soldi, F. Autieri, and S. Vinco, "Automatic integration of SystemC in the FMI standard for software-defined vehicle design," in *Proc. of FDL*, 2025, pp. 1–9.

[25] A. M. Albu, G. Pollo, A. Burrello, D. J. Pagliari, C. Tesconi, L. Panaro, D. Soldi, F. Autieri, and S. Vinco, "Integrating SystemC TLM into FMI 3.0 co-simulations with an open-source approach," in *Proc. of DVCon Europe*, 2025, pp. 29–35.

[26] T.-C. Yeh and M.-C. Chiang, "On the interface between QEMU and SystemC for hardware modeling," in *Proc. of ISNE*, 2010, pp. 73–76.

[27] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 593–606, 2011.

[28] B. Kleinert, S. Weiß, F. Schäfer, J. Bakakeu, and D. Fey, "Adaptive synchronization interface for hardware-software co-simulation based on SystemC and QEMU," in *Proc. of SIMUTOOLS*, 2016, p. 28–36.

[29] I.-H. Chen, C.-T. King, Y.-H. Chen, and J.-M. Lu, "Full system emulation of embedded heterogeneous multicores based on qemu," in *Proc. of IEEE ICPADS*, 2018, pp. 771–778.

[30] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego, "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0," in *Proc. ERTS*, 2016.

[31] L. Jünger, J. H. Weinstock, and T. Kraus, "Scalable software testing in fast virtual platforms: Leveraging SystemC, QEMU and containerization," 2025.

[32] T. Kraus, A. Sauer, and I. Feldner, "Deployment of containerized simulations in an API-driven distributed infrastructure," 2025.

[33] L. Ochel, R. Braun, B. Thiele, A. Asghar, L. Rogovchenko-Buffoni, M. Eek, P. Fritzson, D. Fritzson, S. Horkeby, R. Hällqvist, A. Kinnander, A. Palanisamy, A. Pop, and M. Sjölund, "OMSimulator - Integrated FMI and TLM-based Co-simulation with Composite Model Editing and SSP," in *Proc. of Modelica Conference*, 02 2019, pp. 69–78.

[34] H. Kim, H. Lee, and J. Cho, "A fmi-based approach for CAN bus simulation and simulink model integration in vehicle simulation environment," in *Proc. of ICTC*, 2023, pp. 279–284.

[35] M. Krammer, H. Martin, Z. Radmilovic, S. Erker, and M. Karner, "Standard compliant co-simulation models for verification of automotive embedded systems," in *Proc. of FDL*, 2015, pp. 1–8.

[36] W. Muller and E. Widl, "Using FMI components in discrete event systems," in *Proc. of MSCPES*, 2015, pp. 1–6.

[37] S. Ravi, L. Beermann, O. Kotte, P. Pazzaglia, M. Vinnakota, D. Ziegenbein, and A. Hamann, "Timing-aware software-in-the-loop simulation of automotive applications with FMI 3.0," in *Proc. of ACM/IEEE MODELS*, 2023, pp. 62–72.

[38] N. Bosbach, M. Schmidt, L. Jünger, M. Berthold, and R. Leupers, "FMI meets SystemC: A framework for cross-tool virtual prototyping," in *Proc. of International Modelica & FMI Conference*, vol. 218, Oct. 2025.

[39] A. M. Albu, "Sofia: Automated fmi 3.0 toolchain for wrapping systemc (rtl/tlm/ams) into fmus and integrating them with qemu-based virtual hardware-in-the-loop (vhil) platforms," https://github.com/andreialbu28/SOFIA, 2026, gitHub repository.

[40] A. M. Kaushik, Z. Wu, and H. Patel, "systemc-clang," https://systemc-clang.readthedocs.io, 2020.



Andrei Mihai Albu received the M.Sc. degree in Mechatronic Engineering from Politecnico di Torino, Turin, Italy, in 2024. He is pursuing the Ph.D. degree in Computer and Systems Engineering at Politecnico di Torino. His research interests include virtual prototyping and system-level modeling.



Dario Soldi received the Ph.D. degree in Physics and Astrophysics from the University of Turin, Turin, Italy. He is a Prototyping Engineer at Dumarey Softronix, Turin, Italy. His expertise includes embedded systems, firmware, FPGA-based systems, and automotive software.



Fabio Autieri is an Engineer at Dumarey Group, Turin, Italy. His activities focus on automotive technology and software-defined vehicle development.



Enrico Macii (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Politecnico di Torino in 1995. He is a Full Professor of Computer Engineering with Politecnico di Torino. His research interests are in the design of digital electronic circuits and systems, with a particular emphasis on low-power consumption aspects, energy efficiency, sustainable urban mobility, and clean and intelligent manufacturing.



Sara Vinco (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2013. She has been an Associate Professor with Politecnico di Torino, Turin, Italy, since 2021. Her research interests include digital twins, energy efficient electronic design automation, efficient simulation and optimization of energy systems, and virtual prototyping of heterogeneous embedded systems. She is an Associate Editor of IEEE TCAD.