POLITECNICO DI TORINO
Repository ISTITUZIONALE

Automatic integration of SystemC in the FMI standard for Software-defined Vehicle design

Original
Automatic integration of SystemC in the FMI standard for Software-defined Vehicle design / Pollo, G., Albu, A.M., Burrello, A., Pagliari, D.J., Tesconi, C., Panaro, L., Soldi, D., Autieri, F., Vinco, S.. - (2025), pp. 1-9. (Forum on specification & Design Languages (FDL) 2025 St. Goar (DEU) 10-12 September 2025) [10.1109/fdl68117.2025.11165273].

Availability:
This version is available at: 11583/3003693 since: 2025-10-06T13:14:37Z

Publisher:
IEEE

Published
DOI:10.1109/fdl68117.2025.11165273

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

©2025 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)

27 November 2026

Automatic integration of SystemC in the FMI standard for Software-defined Vehicle design

Giovanni Pollo[1], Andrei Mihai Albu[1], Alessio Burrello[1], Daniele Jahier Pagliari[1],
Cristian Tesconi[2], Loris Panaro[2], Dario Soldi[2], Fabio Autieri[2], Sara Vinco[1]
[1] Politecnico di Torino, Italy - [2] Dumarey Group, Italy
Emails: name.surname@polito.it

arXiv:2508.19665v1 [cs.CL] 27 Aug 2025

***Abstract*—The recent advancements of the automotive sector demand robust co-simulation methodologies that enable early validation and seamless integration across hardware and software domains. However, the lack of standardized interfaces and the dominance of proprietary simulation platforms pose significant challenges to collaboration, scalability, and IP protection. To address these limitations, this paper presents an approach for automatically wrapping SystemC models by using the Functional Mock-up Interface (FMI) standard. This method combines the modeling accuracy and fast time-to-market of SystemC with the interoperability and encapsulation benefits of FMI, enabling secure and portable integration of embedded components into co-simulation workflows. We validate the proposed methodology on real-world case studies, demonstrating its effectiveness with complex designs.**

***Index Terms*—SystemC, Functional Mockup Unit, Functional Mockup Interface, Software-defined Vehicle**

I. Introduction

The automotive industry is undergoing a significant transformation, driven by the rapid development of electric vehicles (EVs), autonomous driving systems, and increasingly sophisticated driver assistance features [1]. These changes impact the complexity of vehicle architectures, and require robust design methodologies that allow for early validation, rapid iteration, and reliable system integration [2].

In parallel, the industry is embracing the Software-Defined Vehicle (SDV) paradigm, where software functionality plays a central role in vehicle capabilities [3], [4]. This transformation requires comprehensive system validation strategies [5], [6]. In this context, co-simulation is crucial, as it enables:

- the joint simulation of different subsystems, when the physical prototypes are not yet available;
- the analysis of complex systems under realistic operating conditions, with each subsystem being simulated in its native simulation environment;
- the synchronized simulation of software modules alongside physical components and control hardware, ensuring that embedded software meets performance and safety requirements under diverse operating conditions [7].

To ensure the effectiveness of co-simulation in large, multidisciplinary projects, standardization of simulation interfaces is crucial, as a mean to promote tool interoperability, improve scalability, and enhance maintainability throughout the development lifecycle. Additionally, the increasing role of collaborative development among original equipment manufacturers, suppliers, and software vendors, implies a growing need to protect Intellectual Property (IP), as sharing detailed source models between parties often poses confidentiality and competitive risks. Therefore, simulation approaches that encapsulate models, while allowing at the same time their simulation, are highly desirable.

Many existing co-simulation solutions are proprietary platforms [8], [9]. These platforms often come with limitations regarding licensing, extensibility, and interoperability. To address these limitations, there is increasing momentum toward open-source solutions that offer transparency, customizability, and a lower entry barrier.

In this context, the Functional Mock-up Interface (FMI) standard has gained wide adoption [10]. FMI defines a tool-agnostic interface for exchanging and co-simulating dynamic systems. The standard allows models to be exported as packaged containers encapsulating compiled code, metadata, and interface specifications. This makes FMI a promising solution for scenarios that require standardized communication and safe IP model sharing. FMI currently supports many languages and frameworks in the context of automotive design flows, including Modelica, Simulink, and Dymola [11].

The SDV paradigm shift is on the other hand increasing the relevance of hardware/software co-design methodologies and languages. In particular, SystemC has established itself as a leading modeling language for embedded systems and hardware architectures [12], [13], as its event-driven simulation capabilities and high-level abstraction make it well-suited for modeling complex digital systems and their timing behavior. SystemC models are often used to describe components such as ECUs, sensors, and controllers, which are central to modern vehicle platforms [14], [15]. However, SystemC lacks native support for standardized co-simulation interfaces, making its integration into broader system simulations challenging. Some research efforts tried to integrate SystemC in the FMI flows, but with limitations in terms of support for synchronization, interrupt modeling, and automation [16]–[24].

To bridge this gap, we propose an approach that *automatically wraps SystemC models by using the FMI standard*. This hybrid solution combines the modeling precision of SystemC with the interoperability and encapsulation benefits of FMI. By doing so, we aim at creating a portable and secure modeling interface that can be easily integrated into larger co-simulation environments, still supporting IP protection and providing a standardized interface for simulation.

The paper is structured as follows: Section II provides the necessary background and state of the art. Section III focuses on the methodology and its automation, that is then applied

©2025 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 collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Accepted for publications at FDL 2025 Forum on specification & Design Languages.

to case studies in Section IV. Finally, Section V draws our concluding remarks.

II. Background and Related Works

A. Functional Mock-up Interface

FMI is a standardized, open interface that facilitates the exchange and co-simulation of dynamic models across different simulation tools [10]. It enables the encapsulation of models into self-contained components called Functional Mock-up Units (FMUs), which can be easily shared, reused, and integrated across various platforms without exposing proprietary information or requiring access to the original source code and modeling environment.

FMUs execute according to three different solutions: Model Exchange (all FMUs are described as systems of equations, solved by a shared solver), Scheduled Execution (FMUs share a common external scheduling algorithm), and Co-Simulation (each FMU incorporates its own solver and autonomously progresses its internal state according to specified communication intervals). In the automotive context, given the heterogeneity of the domains to be covered, Co-Simulation is the most frequent choice, and is thus the target of this work.

An FMU is structured as a compressed archive (`.fmu` file) containing several key elements:

- An XML-based model description file (`modelDescription.xml`) that defines variables, parameters, input/output signals, data types, and model structure.
- Platform-specific binary files (shared libraries, e.g., `.dll`, `.so`) that implement the simulation logic.
- Optional resources such as documentation, initialization files, lookup tables, or graphics.

The FMI standard defines an extensive set of function calls to support consistent and flexible operation across simulation tools. Full specifications are available here [10]; for simplicity, only a subset of these functions is listed below:

- `fmi3Instantiate`: creates a new instance of an FMU;
- `fmi3SetXXX`: function to set the value of a variable, where `XXX` is one of the FMI data types (e.g., `Int8`, `Int32`, `Int64`, `Uint`, `String`, `Boolean`);
- `fmi3DoStep`: advances simulation by a specific time interval;
- `fmi3GetXXX`: function to retrieve the value of a variable, where `XXX` is one of the FMI data types;
- `fmi3FreeInstance`: releases resources allocated to an FMU instance when no longer needed.

B. SystemC

SystemC is a powerful C++ class library that extends standard C++ with hardware modeling capabilities, enabling system-level design and verification of complex digital systems [12]. It provides a unified framework where both hardware and software components can be described, simulated, and verified together at various level of abstraction, from high-level functional models to detailed Register-Transfer Level (RTL) implementations. As its core, SystemC consists of:

- A discrete time, event-based simulation kernel that schedules and executes processes on the occurrence of synchronizations, time notifications or signal value changes, with specific functions to start (`sc_start()`) and pause (`sc_pause()`) simulation and to dynamically generate new processes whenever needed (`sc_spawn()`);
- A potentially hierarchical organization of modules (`SC_MODULE`), which encapsulate functionality in processes, and communicate via ports and channels.
- Primitive channels and interfaces for communication between components.
- Hardware-oriented language constructs, including specific data types, such as bit vectors and logic values (e.g., `sc_bit`, `sc_bv`, `sc_logic`, `sc_lv`).

This paper focuses specifically on SystemC RTL, where simulation relies on cycle-accurate timing models to reflect hardware clock behavior, and on a detailed signal semantics to capture transitions and propagation delays through combinatorial logic. As a result, SystemC offers a versatile and efficient platform for modeling and simulating hardware systems with high fidelity, making it well-suited for both early-stage design exploration and detailed implementation.

C. Related Works

Ever since its first release in 2010, the FMI standard has garnered significant attention from both academia and the automotive industry [25], that has led to its widespread adoption across various simulation environments.

In [16], the authors propose one of the earliest attempts to integrate FMI for Model Exchange into discrete event systems, where FMI components generated with OpenModelica are embedded within the discrete event domain of Ptolemy II. More recently, [17] presents an FMI 2.0-based approach for integrating CAN bus simulation into Simulink-based vehicle simulations, including realistic CAN communication between various vehicle subsystems, message transmission timing and priority-based arbitration. In [18], a modular co-simulation architecture for timing-aware Software-in-the-Loop (SiL) simulation of automotive applications using the FMI 3.0 standard is proposed. Their approach couples timing simulation with functional simulation, allowing for early evaluation of software behavior on target hardware. The architecture consists of a timing simulator, virtual ECU, communication point service, and mode service, each implemented as a Co-Simulation FMU. The case study demonstrates how timing-aware simulations can reveal functional behavior deviations that are not captured by traditional SiL testing, potentially reducing reliance on Hardware-in-the-Loop validation.

However, none of the approaches mentioned above address the integration of SystemC, or any other hardware description language, within the FMI standard. One attempt to bridge this gap is presented in [19], where the authors use FMI 1.0 to co-simulate a hardware model specified at the RTL level using the DEVS Suite Simulator with a software model defined as a MATLAB script. [21] tackled the integration challenge by proposing a method to incorporate SystemC-based virtual prototypes into heterogeneous, multi-domain automotive simulations via the FMI standard. However, their solution

TABLE I
RELATED WORKS

	FMI VERSION	SYSTEMC SUPPORT	METHODOLOGY AND TOOLS
[16]	2.0	✗	OpenModelica and Ptolemy II required
[17]	3.0	✗	Custom for CAN bus
[18]	3.0	✗	No tools required
[19]	1.0	≈	DEVS and Matlab required
[20]	2.0	✓	No interrupts
[21]	2.0	✓	Platform Architect required
[22]	2.0	✓	Overhead in the integration
[23]	2.0	✓	Overhead in the integration
[24]	2.0	✓	Requires VPSim
Our	(1.0) (2.0) 3.0	✓	No tools required Simple Integration Full Interrupt support

relies heavily on Platform Architect, a commercial tool from Synopsys for assembling virtual prototypes [26]. In [22], a methodology is proposed for integrating both SystemC and SystemC/AMS models into FMUs using FMI 2.0. While the method is general, it requires the addition of external functions outside of the FMI specification to manage SystemC execution. [20] introduced a bridge between SystemC and FMI 2.0 to enable exporting SystemC models as FMUs. While effective, the authors identified limitations in FMI's native support for event-driven communication and data types. Additionally, interrupt handling was not considered in their work. In [23], a generic and configurable FMI master is developed to facilitate communication between SystemC/TLM-based virtual platforms and FMUs from various tools. This approach supports multiple FMU slaves but necessitates modifying the SystemC design, as the FMI master is instantiated within the top-level module encapsulating the SoC model. Lastly, [24] presents a general and scalable methodology for co-simulating Cyber-Physical Systems (CPS) using FMI. However, the method is closely tied to the VPSim environment for virtual prototyping [27], thus limiting its portability.

An overview of the related work discussed above is provided in Table I. In summary, existing methods either depend on proprietary tools, lack seamless integration with the FMI standard, or require modifications to the SystemC design. Our proposed approach addresses these limitations by offering a fully open-source, simple, and extensible solution to integrate any SystemC design into an FMU.

III. METHODOLOGY

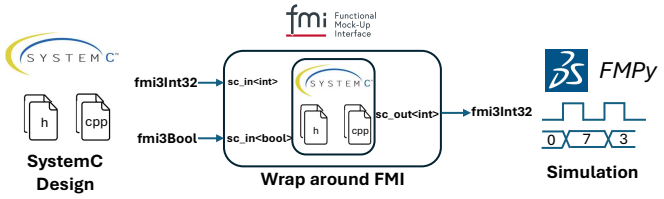


Fig. 1. Methodology Overview in three steps: input SystemC Design, generation of the FMI wrapper and Simulation driven by a FMI master

This section provides a detailed explanation of the methodology, starting with a high-level overview (Figure 1) and progressively diving into each specific detail. Finally, it presents the automation framework, which integrates all methodological aspects in a plug-and-play fashion, making the use of the tool almost transparent to the end user.

A. High Level Overview

The methodology proposed in this paper is a three-step flow, as summarized in Figure 1. From left to right, the first step is the selection of the SystemC design, that is the entry point of the flow. In the second step, the SystemC design is encapsulated in a FMI wrapper, that ensures a standard co-simulation interface with no internal modifications or adaptations of the design itself. This phase allows to map input and output ports to corresponding FMI data types, to implement the necessary FMI-compliant functions, and to generate the resulting `.fmu` file. Finally, the flow concludes with the simulation of the design, performed by using a master that invokes the FMI functions to control the evolution of simulation. The master can be either implemented with an open library, e.g., FMPy [28], or within proprietary tools, thus extending the applicability of the methodology to commercial environments.

B. SystemC Design Pre-Processing

The entry point of the methodology is the initial SystemC design, that must be wrappped as a FMU. The proposed methodology does not impose any limitation on the chosen design and supports all designs written in SystemC at the RTL level. The only assumption is that the design contains a top level entity with input/output ports to be mapped onto FMU variables, and that the source code of the interface is accessible to extrapolate the list of such ports. Such design may be hierarchical, i.e., contain a number of SystemC modules. Given that the proposed approach does not require source code modifications, any SystemC version and any design for which the source code is accessible can be directly integrated.

The top-level entity of the SystemC design is parsed to extrapolate information necessary to define the `ModelDescription.xml` file [10], that contains all necessary metadata for the FMU (e.g., the FMI version and the model name), as well as a complete description of the SystemC interface. The SystemC design is thus parsed, to extrapolate the complete list of ports, complete of their name, direction (input/output), type.

C. FMI wrapper construction

The next step consists of wrapping the SystemC design with the necessary FMI 3.0 constructs. This wrapping is implemented as a container that includes both SystemC and FMI components, to bridge the two domains, necessary to allow and control SystemC simulation, and to make the values of SystemC ports accessible from the FMI functions, to set the correct inputs and gather the corresponding outputs. This is achieved with the declaration of a `struct` containing a reference to an instance of the SystemC top level entity (allocated at run time), plus one FMI variable and one SystemC signal for each SystemC input and output port (middle diagram of Figure 1):

- the signals will be used to bind the ports, as all top-level entity ports must be bound to either ports or signals to

TABLE II
MAPPING OF THE PORTS BETWEEN SYSTEMC AND FMI

SYSTEMC DATA TYPE	FMI DATA TYPE
sc_logic	fmi3Bool
sc_bv<N>	fmi3Binary
sc_int<1 .. 8>	fmi3Int8
sc_int<9 .. 16>	fmi3Int16
sc_int<17 .. 32>	fmi3Int32
sc_int<33 .. 64>	fmi3Int64
sc_uint<1 .. 8>	fmi3UInt8
sc_uint<9 .. 16>	fmi3UInt16
sc_uint<17 .. 32>	fmi3UInt32
sc_uint<33 .. 64>	fmi3UInt64
sc_float	fmi3Float32
sc_double	fmi3Float64

allow successful SystemC simulation. The signals have the same type as the original SystemC port, and their name is the same as the port, with a `s_` suffix;

- the variables will be used to store the corresponding values in an FMI-compatible format. The name of the variables is the same as the original SystemC port. Each port type is mapped onto an FMI type, as summarized in Table II. In addition to SystemC-specific types, all standard C++ data types are also supported and mapped accordingly (e.g., `int` is mapped to `fmi3Int32`, `float` to `fmi3Float32`, etc.). This mapping is designed to be extensible: any new types introduced in future FMI versions will be coherently mapped to the closest matching C++/SystemC data types.

The next step involves implementing the prototype functions required by the FMI standard. These functions define the core interactions between the simulation environment and the SystemC design:

- the `fmi3InstantiateCoSimulation` function instantiates the SystemC top level design and binds the SystemC signals to the design ports. At this point, any further initialization can be applied (e.g., to set default values);
- the `fmi3SetXXX` and `fmi3GetXXX` functions are then used to map values of the FMI variables onto the SystemC signals and vice versa. This ensures data passing between the FMI domain and the SystemC domain, to give the correct inputs throughout the simulation and gather the corresponding results;
- the `fmi3DoStep` is used to manage the progression of SystemC simulation, as will be detailed in the next section;
- the `fmi3FreeInstance` function frees any dynamic allocated memory, and invokes the destructor of the SystemC top level entity.

After the complete set of prototype functions has been developed and verified, the design becomes ready for compilation. According to the FMI standard, a platform-specific compilation is expected to produce an executable library, e.g., a Dynamic Link Library (`.dll`) for Windows or a shared object library (`.so`) for Linux systems.

D. FMI-controlled SystemC Simulation

The activation of SystemC simulation must respect both the SystemC execution semantics (without requiring modifications to the design and/or to the simulation kernel) and the FMI simulation structure. At the same time, timing plays a crucial role in the RTL simulation paradigm, and it must thus be properly synchronized between the two environments.

The FMI standard allows three main types of execution:

a) Step mode: The first mode activates the FMUs at fixed time steps. In this scenario, the `ModelDescription.xml` file contains a `CommunicationStepSize` parameter, that conveys the size of the fixed time step to be used at any FMU invocation. To synchronize the SystemC simulation with this mechanism, an invocation of the `fmi3DoStep` internally triggers SystemC simulation with an invocation of the `sc_start()` for a duration corresponding to the specified `CommunicationStepSize`. An example of this mechanism is visible in Figure 2, where each `fmi3DoStep(15ms)` triggers the `sc_start(15ms)`. This straightforward encapsulation ensures synchronization between the SystemC simulation and the FMI co-simulation environment without requiring any modification to the SystemC simulation kernel.

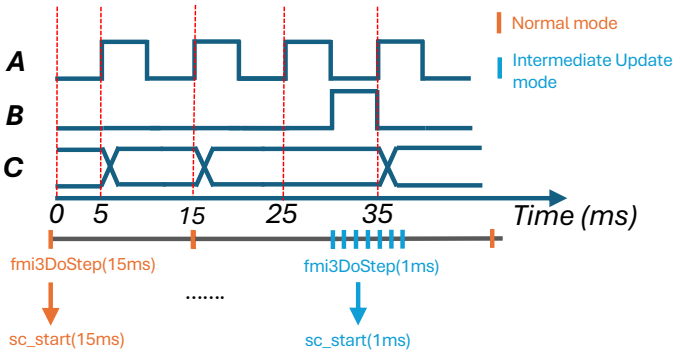


Fig. 2. Step (orange) and intermediate update (blue) modes.

b) Intermediate update: This second mechanism is provided by the FMI standard to enable the exchange of input and output values between communication points. This is particularly useful in scenarios where a finer simulation granularity is required (e.g., in case of physical simulations). When the FMU `IntermediateUpdateMode` is detected, the framework dynamically reduces the `CommunicationStepSize` used by the `fmi3DoStep`, to advance SystemC simulation time with finer temporal resolution. A visualization of this mechanism is represented in Figure 2, where at a certain point of the simulation, the `CommunicationStepSize` becomes of 1ms (blue dashes), exploiting a finer grain execution.

c) Interrupt management: The final and arguably most critical timing mechanism is the event-based or interrupt-driven execution. Handling interrupts requires a more sophisticated management approach, as an interrupt typically triggers the execution of an Interrupt Service Routine (ISR). Two solutions have been developed to address this need, differing mainly in their invasiveness and complexity. Both solutions are illustrated in Figure 3, where B is the signal treated as interrupt.

The first approach, which is capable of handling asynchronous interrupts, requires a minimal adaptation of the SystemC wrapper around FMI, although it does not necessitate any modifications to the original SystemC design. The core

idea is to generate a dynamic SystemC process (with the `sc_spawn()` primitive), used to detect when a specified condition is met (e.g., the rising edge of the signal interrupt). Upon detecting such condition, the system enters the FMI `EventMode` [29] and calls the `sc_pause()` primitive, that pauses the SystemC simulation and the execution of the `fmi3DoStep`. The ISR is then executed, after which the system returns to FMI `StepMode` [30], thus resuming normal simulation until the next interrupt occurs. This allows handling asynchronous interrupts with high precision.

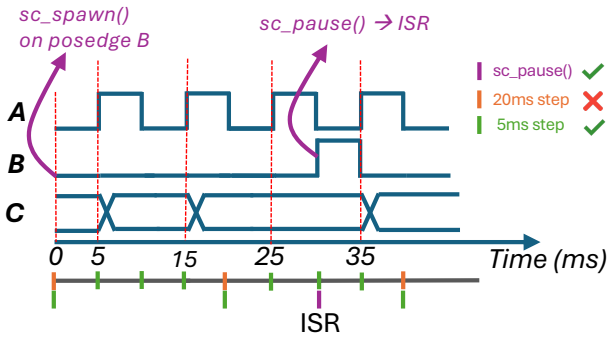


Fig. 3. Event-based/Interrupt-driven mode with the two proposed solutions.

The second approach is engineered to avoid any modifications to the internal simulation code, thus maximizing portability and simplicity. However, it is limited to managing synchronous interrupts and its accuracy depends on the simulation step size. In this case, one or more signals are selected for monitoring, and at the end of each `fmi3DoStep`, their values are checked against the desired interrupt condition. If the condition is met, the ISR is executed. The limitation of this solution is clearly that its ability to detect interrupts is directly tied to the simulation's temporal resolution. In Figure 3, the finer step size ($5ms$, green) allows for successfully detection of the interrupt, while the larger step size ($20ms$, orange), misses the interrupt. A correct sizing of the `CommunicationStepSize` size parameter is thus crucial. On the other hand, this solution offers a good trade-off between complexity and simulation accuracy when a sufficiently fine simulation step is adopted. This trade-off is an intentional design choice aimed at maximizing portability and simplicity, specifically to avoid any modifications to the SystemC wrapper. While this may come at the cost of reduced accuracy, it enables a non-invasive integration strategy that remains broadly compatible and easier to adopt.

E. Simulation using FMPy

The final stage of the proposed workflow consists of simulating the generated FMU (rightmost block of Figure 1). For this purpose, the FMPy Python library was selected. FMPy simplifies the simulation process by exposing high-level APIs that internally call the standard FMI functions, facilitating seamless execution [28]. The library offers full support for all functions defined in the FMI 3.0 specification and is compatible with each of the execution modes described in the previous sections. However, it is important to note that the generated FMU can be integrated in any FMU-based environment.

F. Automation Framework

The previous sections have primarily focused on the methodology and internal mechanisms of the proposed framework, with limited attention given to implementation-specific details. This separation is intentional, as all previously described processes have been automated to maximize usability and accessibility. To this end, a lightweight yet extensible automation framework has been developed, aimed at streamlining FMU generation and simulation with minimal user intervention. An overview is reported in Figure 4.

The automation process takes as input two elements: the SystemC design and a configuration file written in YAML format. The YAML file serves multiple roles, including defining the paths to various SystemC source files and specifying configuration parameters for FMU generation. It also simplifies the definition of the FMU's XML metadata by abstracting the verbosity of the `ModelDescription.xml` into a minimal set of user-defined fields. Additional configuration parameters include simulation-related attributes such as step size and total simulation duration.

Once configured, the automation is triggered by executing a Python script that carries out the following steps:

1) **Parsing** - Analyzes the SystemC design to identify its input and output ports.
2) **Model Description Generation** - Creates the `ModelDescription.xml` file using both the parsed port information and user-provided configuration.
3) **FMI Function Implementation** - Generates all required FMI functions, incorporating the appropriate port mappings described in Section II.
4) **Compilation** - Compiles both the SystemC design and the generated FMI glue code using either a Bash script or CMake, depending on user preference.
5) **FMU Packaging** - Packages the compiled binaries into a platform-specific FMU, producing a `.dll` for Windows or a `.so` for Linux, thus supporting both Operating Systems.
6) **Simulation Execution** - Launches a simulation run using the newly created FMU.

This automation pipeline enables a nearly hands-free workflow, significantly reducing the barrier to entry for users who need to encapsulate SystemC designs into FMUs for co-simulation purposes. The framework is available at `https://github.com/eml-eda/systemc-fmi`.

IV. Experimental Results

To evaluate the effectiveness of the proposed integration method, we conducted an extensive series of experiments using a set of heterogeneous SystemC models, including custom-developed examples, industrial case studies by partner companies, and publicly available models sourced from GitHub. Each case study underwent the integration process outlined in Figure 4, after which we carried out several performance assessments. Experiments were conducted on Ubuntu 24.04 LTS running inside WSL, on a host machine equipped with an Intel i7-12700H processor and 16GB of RAM. Specifically, we measured and reported in Table III:

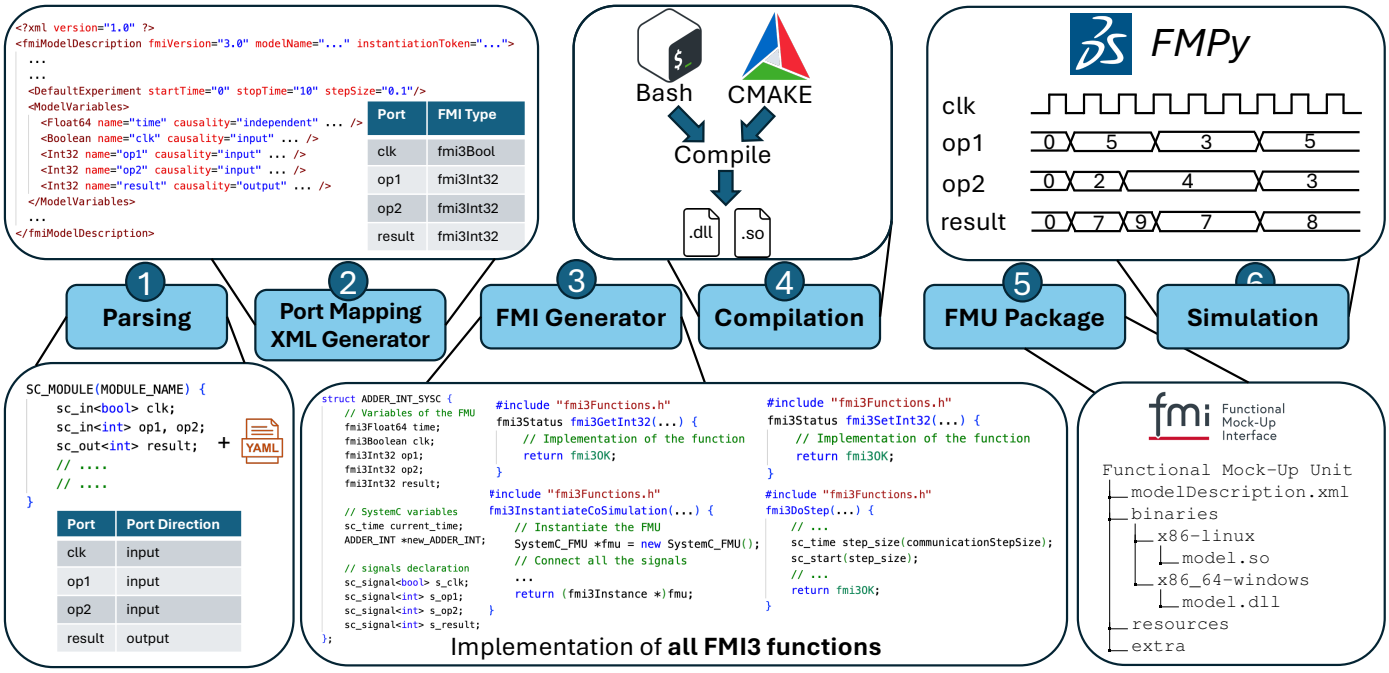


Fig. 4. Automation Framework exemplified on the ALU case study.

- *Model complexity*, evaluated based on the number of: ports, processes, lines of code (LoC), and of hierarchical modules, that should give a measure of the size and complexity of each case study;
- *Total simulation time*, comparing native SystemC execution to simulation as FMU controlled via the FMPy library. The profiling infrastructure for the native SystemC version was developed in C++ (s columns), whereas the FMU was assessed using the Python-based co-simulation framework using libraries such as `psutil` [31] and is reported as slow down factor, i.e., ratio between the FMU-based and native SystemC execution times ($\times$ columns). Each experiment was executed five times, and the average of the results was computed to enhance the reliability of the collected data. Both the encapsulated FMU model and the native SystemC implementation were evaluated under varying simulation durations, ranging from 100 ms to 10s, and the step size of FMU execution was set to 1 ms for all experiments;
- *Memory overhead*, reported as SystemC memory request (MB columns) and FMI-induced overhead ($\times$ columns).

A. Arithmetic Logic Unit

As an introductory example to illustrate the overall integration workflow in Figure 4, we selected a SystemC design of an Arithmetic Logic Unit (ALU) containing a single synchronous process featuring three unsigned integer inputs (two 4-bit operands and a 3-bit opcode with 8 different operations) and a single 4-bit unsigned integer output representing the result. As discussed in Table II, these data types are mapped to `fmi3UInt8` variables in the `modelDescription.xml file`, serving as input and output representations in the FMU.

After encapsulating the SystemC model as a FMU, a comparative evaluation was performed against the native SystemC implementation. As expected, the FMU-based simulation introduces additional performance overhead due to the abstraction layers and runtime interfacing inherent in the FMI framework. The slowdown factor, defined as the ratio between the FMU-based and native SystemC execution times, goes from approx. $16\times$ with the shortest simulation to $12\times$ for the longest one. This decreasing trend highlights the presence of a fixed overhead introduced by the FMI infrastructure, mostly associated with one-time operations such as `fmi3InstantiateCoSimulation`, `fmi3SetupExperiment`, and `fmi3EnterInitializationMode`, executed only during the setup phase.

In terms of memory usage, the native SystemC simulation consumed approximately 4.6MB in all simulation scenarios, while the encapsulated FMU version increased memory consumption from approx. $24.5\times$ to $28\times$. This rise in memory usage can be attributed to the FMI runtime, that requires operations like data translation and conversion. While the memory overhead appears high, especially for such a simple model, it is important to recognize that such cost stems mainly from the flexible, standardized infrastructure defined by the FMI specification, rather than the inherent complexity of the model itself, and by the adoption of a Python-based master, inherently less efficient than a full C++-based implementation. Crucially, the memory usage remains stable across runs and scales predictably, making it acceptable in workflows where modularity, tool interoperability, or platform independence is prioritized over minimal resource consumption.

B. Cyclic Redundancy Check (CRC)

The Cyclic Redundancy Check (CRC) case study was developed in collaboration with our industry partner for application in automotive verification, where configurable CRC units are critical for error detection in communication protocols such as

TABLE III
EXPERIMENTAL RESULTS

	DESIGN CHARACTERISTICS				EXECUTION TIME						MEMORY					
	PORTS (#)	PROCESSES (#)	MODULES (#)	LoC (#)	100 ms		1 s		10 s		100 ms		1 s		10 s	
					s	×	s	×	s	×	MB	×	MB	×	MB	×
ALU	4	1	1	317	0.003	15.98	0.012	15.16	1.421	12.08	4.650	24.53	4.620	25.03	4.650	27.79
CRC	12	15	1	2,467	0.021	4.07	0.082	2.30	0.642	1.84	5.530	19.91	5.560	19.92	5.510	20.93
I2C	10	11	2	1,085	0.022	8.95	0.134	9.53	2.362	6.28	5.547	18.75	5.652	18.76	7.336	17.98
RISC	32	8	6	731	0.013	11.78	0.029	12.34	0.138	14.87	5.280	21.57	5.332	21.61	5.036	21.78
Delta Sigma	4	1	1	354	0.033	14.85	0.330	4.38	3.283	3.61	4.430	23.12	4.434	23.21	5.270	20.30

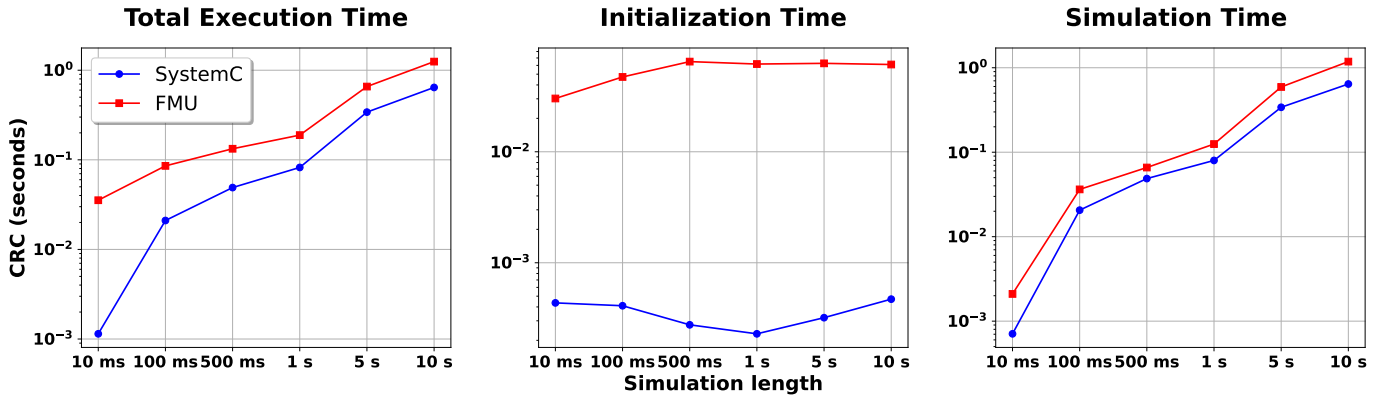


Fig. 5. CRC simulation performance: SystemC (blue) and FMI-wrapped SystemC (red). Initialization time is almost constant for both SystemC and FMI across all simulation lengths, while the simulation times scales and dominates the total execution time.

CAN and Ethernet. This CRC module is a high-performance, configurable hardware model intended for integration into larger System-on-Chip (SoC) verification environments supporting 16 independent CRC contexts, each configurable for either CRC-16 or CRC-32 polynomial operations. Additional features include input/output inversion, byte-swapping, register-based control, error detection capabilities, and support for multi-cycle operation handling. The design incorporates five distinct dataflows and three supporting functions, implemented through a total of 15 processes.

The results presented in Figure 5 highlight the trade-offs in timing and memory performance between the native SystemC model and the encapsulated one. The slowdown factor is lower than for the ALU, as the CRC case study includes more computation, that compensates for the FMI-induced overhead. The slowdown factor goes thus from approx. $4\times$ to less than $2\times$, when increasing simulation length. Also in this case there is a fixed overhead introduced by the FMI infrastructure, associated to one-time initialization operations, highlighted by Figure 5: initialization time is almost constant for both SystemC and FMI across all simulation lengths, while the simulation times scales and dominates the total execution time.

In terms of memory consumption, SystemC maintains consistent and minimal memory usage across all test cases of around 5.5MB. Conversely, FMU requires around $20\times$ more memory, in particular in terms of Resident Set Size (RSS, i.e., RAM memory, $20\times$) and of Virtual Memory size (VM, from $10\times$ to almost two orders of magnitude in worst case).

C. I2C (Inter Integrated Circuit)

As a communication-oriented case study, we adopted an open-source I2C bus model, typical of automotive systems [32]. This case study is hierarchical, i.e., includes two SystemC modules. The master supports core I2C functions ($START/STOP$ conditions, 7-bit addressing, data read/write, and $ACK/NACK$ signaling) via a finite state machine managing protocol phases. The slave responds to address $0x2A$, decoding commands and issuing responses such as data output or write acknowledgments. Both modules manage bidirectional SDA and SCL lines using tri-state control for arbitration, ensuring protocol compliance, clock synchronization, signal integrity. The module supports basic error detection through the $NACK$ signal, set to 1 for invalid addresses.

The I2C case study is wrapped as a FMU by treating the $NACK$ signal as an interrupt through the `EventMode` approach (Section III-D.c). Simulation profiling shows a slowdown factor that scales from $9\times$ for longest simulations to approx. $6\times$ for shortest ones. The higher overhead w.r.t. the former case study can be explained from the need to reinitialize the FMU at each invocation of the `doStep` function. This overhead increases from 100ms to 1ms long simulations, as it requires a lot of variable set operations, and is compensated only with longer simulations, starting from 10s. In this case study, SystemC memory usage increases with time, as an effect of the I2C protocol, but remains lower than 7.5MB. FMI induces an overhead of about $18\times$ ($19\times$ for RSS, $88\times$ for VM, and $7.6\times$ for shared memory).

D. RISC Processor

As computation- and hierarchy-oriented design, we adopted a SystemC-based Reduced Instruction Set Computing (RISC) processor [33]. The RISC CPU core is a pipelined processor designed with five stages, that allow the execution of one instruction per millisecond. It operates within a memory hierarchy and contains an ALU supporting a range of operations including addition, subtraction, logical functions, as well as data movement and shift instructions, all with integrated flag generation. A dedicated control system governs pipeline operation, managing

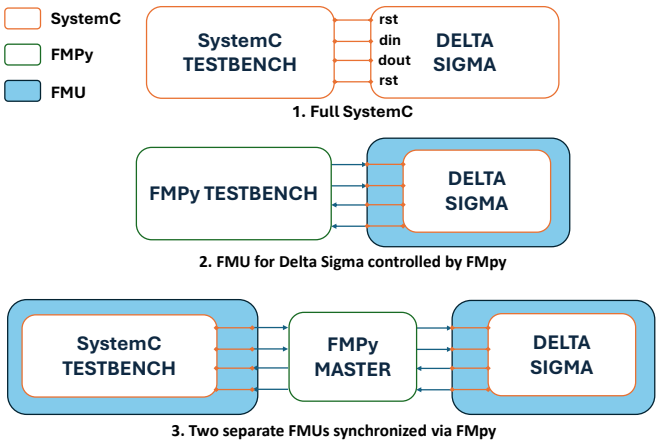


Fig. 6. Configurations compared for the Delta Sigma case study.

instruction flow, resolving hazards, and orchestrating access to both memory and registers via control signals. Execution timing is strictly regulated, with each instruction constrained to a one-millisecond window, enforced by a SystemC clock and monitored through second-by-second progress tracking.

Simulation profiling highlights consistent performance, with a slowdown factor that increases with simulation length, from approx. $12\times$ to $14.5\times$. This is mostly caused by the high number of ports (32), that impose a relatively large overhead at each invocation of the `fmi3SetXXX` and `fmi3GetXXX` functions. The memory overhead is instead stable across all executions (approx. $21.5\times$). These values suggest that the RISC design achieves near-linear scaling w.r.t. simulation duration, implying efficient internal processes and minimal bottlenecks.

E. Delta Sigma Modulator

The last case study is a Delta Sigma modulator provided by an industrial partner. The Delta Sigma converts high-resolution analog input signals into a 1-bit output stream through oversampling and noise shaping. This digital modulator architecture employs n-th cascaded integrator stages that accumulate the difference between the input signal and quantized feedback: at each clock cycle, the system compares the final integrator value against zero to generate the 1-bit output, feeds back its decision to all the n-integrators, and propagates the quantization error through successive integration stages. A synchronous clock generator maintains precise timing with milliseconds resolution, while an active-low reset initializes all integrators to zero. Input is provided by a SystemC testbench, generating a sine wave with variable amplitude and frequency.

The Delta Sigma case study is wrapped into FMU by allowing also the `IntermediateUpdate` mode (Section III-D.b), useful to adjust estimation of the FMU output to the variable frequency and amplitude of the generated input.

In the discussion of this case study, we compare three different configurations, reported in Figure 6. The first experiment is in line with the former experiments, and compares the full SystemC system (Figure 6.1) with the Delta Sigma wrapped as FMU controlled by a FMpy master, feeding the FMU with the same sinusoidal input as the original SystemC testbench (Figure 6.2). Simulation profiling shows a slowdown factor that decreases with the length of simulation, going from approx. $15\times$ to $3.61\times$, as the Delta Sigma is a computation-dominated case study, and compensates well the overhead of initialization and data management induced by FMI. Memory overhead is in the order of $23\times$, similar to the other case studies.

As an additional experiment, we simulated a realistic scenario, where the Delta Sigma is given to a third party as FMU, and the stimuli generation part is a separate FMU interacting via a Python master (Figure 6.3). To achieve this, we also wrapped the original SystemC testbench as a separate FMU. The memory overhead is similar to the former experiment, as the FMI-induced overhead is the same as when executing the Delta Sigma FMU with a Python master (approx. $23\times$). Simulation times show a very interesting trend: the initial overhead for a 100ms simulation is high (0.033s vs 0.440s, approx. $13\times$), but decreases with the 1s long simulation (0.330s vs 0.750s, $2.27\times$) and becomes almost negligible with the 10s long simulation (3.283s vs 3.679s, $1.12\times$). Two considerations arise from these numbers. The first is that the SystemC sine wave generation is more efficient than the Python one, as configuration 3 is more efficient than configuration 2, explored in the former experiment (that for 10s long simulations had a $3.61\times$ overhead). The second consideration is that SystemC case studies wrapped into FMI by the proposed flow can be effectively given to third parties and integrated in external environments, without disclosing the source code and still allowing good simulation performance.

V. Conclusions

In this paper, we proposed an automated framework for integrating SystemC RTL models with the FMI standard, enabling seamless co-simulation between hardware components and software/system-level tools. Our approach facilitates early design space exploration, enhances cross-domain interoperability, and enables third party collaboration.

The framework automatically generates FMUs from SystemC models, exposing standardized interfaces for interaction with FMI-compliant tools. Key advantages include tool independence, allowing broad applicability across simulation environments without proprietary restrictions, and non-invasive integration, that preserves original SystemC models without manual modifications.

The experimental evaluation demonstrated the framework's efficiency across a range of hardware components with differing complexity levels. The proposed framework is capable of attaining integration and standardisation through the usage of FMI standard, at the price of a performance and memory overhead. However, it is important to note that the proposed solution allows to wrap complex and heterogeneous case studies, and to allow smooth integration in third party simulations, without disclosing the source code and avoiding any negative impact on simulation accuracy and effectiveness.

Future work will extend the proposed solution to SystemC TLM, and will analize the integration of the generated SystemC-based FMUs in ISS-based environments (e.g., including QEmu), to prove the integration in the typical hardware-software co-design flow for complex embedded systems.

REFERENCES

[1] E. Yurtsever *et al.*, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020.

[2] N. Crespi, "The digital twin," *The Digital Twin*, vol. 1-2, pp. 1–1238, 1 2023.

[3] Z. Liu *et al.*, "Impact, challenges and prospect of software-defined vehicles," *Automotive Innovation*, vol. 5, pp. 180–194, 4 2022.

[4] U. Bordoloi *et al.*, "Autonomy-driven emerging directions in software-defined vehicles," in *Proc. of DATE*, 2023, pp. 1–6.

[5] Y. Zhou *et al.*, "Hardware software co-design for automotive CPS using architecture analysis and design language," 2016. [Online]. Available: https://arxiv.org/abs/1603.05069

[6] T. Schulze, "Validating software-defined vehicles," *ATZ worldwide 2022 124:6*, vol. 124, pp. 64–64, 5 2022. [Online]. Available: https://link.springer.com/article/10.1007/s38311-022-0822-y

[7] P. Le Marrec *et al.*, "Hardware, software and mechanical cosimulation for automotive applications," in *Proceedings. Ninth International Workshop on Rapid System Prototyping (Cat. No.98TB100237)*, 1998, pp. 202–206.

[8] MATLAB, "Simulink." [Online]. Available: https://it.mathworks.com/products/simulink.html

[9] ETAS, "Powerful co-simulation and integration platform." [Online]. Available: https://www.etas.com/ww/en/products-services/software-development-tools/cosym/

[10] Modelica, "Modelica/fmi-standard: Specification of the functional mock-up interface (fmi)." [Online]. Available: https://fmi-standard.org/

[11] H. Kim *et al.*, "A fmi-based approach for CAN bus simulation and Simulink model integration in vehicle simulation environment," in *Proc. of International Conference on Information and Communication Technology Convergence (ICTC)*, 2023, pp. 279–284.

[12] Accellera, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pp. 1–618, 2023.

[13] V. Muttillo *et al.*, "SystemC-based co-simulation/analysis for system-level hardware/software co-design," *Computers and Electrical Engineering*, vol. 110, p. 108803, 9 2023.

[14] Y. Niimi *et al.*, "Virtual development of engine ECU by modeling technology," *SAE Technical Papers*, 4 2012. [Online]. Available: https://saemobilus.sae.org/papers/virtual-development-engine-ecu-modeling-technology-2012-01-0007

[15] H. Ishihara *et al.*, "Automotive electric system level design methodologies with SystemC," *SAE Technical Papers*, 4 2012. [Online]. Available: https://www.sae.org/publications/technical-papers/content/2012-01-0003/

[16] W. Muller and E. Widl, "Using FMI components in discrete event systems," in *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, 2015, pp. 1–6.

[17] H. Kim *et al.*, "A FMI-based approach for CAN bus simulation and Simulink model integration in vehicle simulation environment," in *2023 14th International Conference on Information and Communication Technology Convergence (ICTC)*, 2023, pp. 279–284.

[18] S. Ravi *et al.*, "Timing-aware software-in-the-loop simulation of automotive applications with FMI 3.0," in *ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2023, pp. 62–72.

[19] M. H. Quraishi *et al.*, "Co-simulation of hardware RTL and software system using FMI," in *Winter Simulation Conference (WSC)*, 2018, pp. 572–583.

[20] S. Centomo *et al.*, "Using SystemC cyber models in an FMI co-simulation environment: Results and proposed FMI enhancements," in *Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 318–325.

[21] R. L. Bucs *et al.*, "Virtual hardware-in-the-loop co-simulation for multi-domain automotive systems via the functional mock-up interface," in *Forum on Specification and Design Languages (FDL)*, 2015, pp. 1–8.

[22] M. Krammer *et al.*, "Standard compliant co-simulation models for verification of automotive embedded systems," in *Forum on Specification and Design Languages (FDL)*, 2015, pp. 1–8.

[23] M. Safar *et al.*, "Virtual electronic control unit as a functional mockup unit for heterogeneous systems," in *Proc. of IEEE ISCAS)*, 2018, pp. 1–5.

[24] S. E. Saidi *et al.*, "Fast virtual prototyping of cyber-physical systems using SystemC and FMI: ADAS use case," in *Proc. of International Workshop on Rapid System Prototyping (RSP'19)*, ser. RSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–49.

[25] T. Blochwitz *et al.*, "The functional mockup interface for tool independent exchange of simulation models," 03 2011, pp. 105–114.

[26] Synopsis, "Platform architect." [Online]. Available: https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html

[27] A. Charif *et al.*, "Fast virtual prototyping for embedded computing systems design and exploration," in *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*, 2019.

[28] "CATIA-Systems/FMPy: Simulate functional mockup units (FMUs) in python." [Online]. Available: https://github.com/CATIA-Systems/FMPy

[29] Modelica, "FMI-standard: FMI Event Mode." [Online]. Available: https://fmi-standard.org/docs/3.0/#EventMode

[30] ——, "FMI-standard: FMI Step Mode." [Online]. Available: https://fmi-standard.org/docs/3.0/#StepMode

[31] psutil, "psutil." [Online]. Available: https://github.com/giampaolo/psutil

[32] H. Boussetta, "hichem/I2C: SystemC design of a master/slave I2C." [Online]. Available: https://github.com/hichem/I2C.git

[33] M. Barrett, "systemc/cpu," 2018. [Online]. Available: https://github.com/michaelbarrett/systemc/tree/master/cpu