Integrating SystemC TLM into FMI 3.0 Co-Simulations with an Open-Source Approach

# Integrating SystemC TLM into FMI 3.0 Co-Simulations with an Open-Source Approach

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



***Abstract*—The growing complexity of cyber-physical systems, particularly in automotive applications, has increased the demand for efficient modeling and cross-domain co-simulation techniques. While SystemC Transaction-Level Modeling (TLM) enables effective hardware/software co-design, its limited interoperability with models from other engineering domains poses integration challenges. This paper presents a fully open-source methodology for integrating SystemC TLM models into Functional Mock-up Interface (FMI)-based co-simulation workflows. By encapsulating SystemC TLM components as FMI 3.0 Co-Simulation Functional Mock-up Units (FMUs), the proposed approach facilitates seamless, standardized integration across heterogeneous simulation environments. We introduce a lightweight open-source toolchain, address key technical challenges such as time synchronization and data exchange, and demonstrate the feasibility and effectiveness of the integration through representative case studies.**

***Index Terms*—Transaction Level-Modeling, SystemC, Functional Mock-up Interface, Software-defined Vehicle**

## I. Introduction

The increasing complexity of embedded and cyber-physical systems in modern vehicles has driven the need for advanced modeling and co-simulation techniques that enable early design exploration, performance evaluation and cross-domain integration [1]. In this context, Transaction-Level Modeling (TLM) using SystemC has emerged as a powerful methodology in Electronic-System-Level (ESL) design, enabling efficient simulation and abstraction for hardware/software co-design and architectural exploration [2], [3].

However, modern systems are highly heterogeneous, and require to take into account multiple domains, including control, thermal, or mechanical [4]. This limitation becomes especially critical in system-level design workflows, where co-simulation involving heterogeneous tools and models is increasingly common. As system integration becomes more multidisciplinary, the need for interoperable, modular simulation standards has become evident [5].

To meet the growing demand for seamless integration in systems engineering, the Functional Mock-up Interface (FMI) standard has gained widespread adoption [6]–[8]. FMI enhances cross-tool interoperability, as it provides a standardized, tool-independent interface to encapsulate simulation models as Functional Mock-up Units (FMUs), which can be shared, reused, and integrated across diverse platforms.

Although FMI is widely used in domains such as control systems and system dynamics (e.g., in Simulink, Dymola, or Modelica-based tools), its adoption in the SystemC TLM ecosystem remains limited. This work investigates and demonstrates the feasibility of integrating SystemC TLM models into FMI-based co-simulation workflows, and proposes a methodology for encapsulating SystemC TLM components as FMI 3.0 Co-Simulation FMUs. This integration is inherently non-intrusive, requiring no modifications to existing TLM models, and achieves full FMI standard compliance with the application of an open source automated tool. Experimental results from two industrial case studies, chosen for their diverse complexity and computation-communication tradeoffs, prove the effectiveness of our FMU integration approach. A final experiment, integrating a generated FMU with a Simulink-based FMU, conclusively proves seamless FMI compatibility.

The rest of the paper is organized as follows: Section II presents background information on SystemC TLM, FMI standard and state of the art; Section III describes the proposed integration method and its automation framework; Section IV presents test studies and experimental results. Finally, Section V concludes the paper and outlines future works.

## II. Background and Related Works

### A. SystemC TLM

SystemC TLM builds on top of the SystemC simulation engine [9], by replacing low-level, pin-accurate signal protocols with higher-level transaction-based communication. This abstraction separates communication from computation, significantly reducing the complexity and improving the efficiency of simulations [10], [11].

In SystemC TLM, a *transaction* represents an operation between components, e.g., a read or write operation, rather than modeling the exact signals and timings on pins or wires. Each transaction is implemented as a *payload*, an object wrapping attributes (e.g., address, data) and protocol phases.

Transactions are exchanged between two entities: the *initiator*, which triggers the operation and owns a `tlm_initiator_socket`, and the *target*, which receives and processes the request via a `tlm_target_socket`. These sockets define the communication interface and support both blocking and non-blocking transport. In blocking transport, the initiator directly calls the target's `b_transport()` method, which processes the request immediately. In non-blocking transport, the initiator issues a request using the `nb_transport_fw()` method; the target acknowledges receipt, and sends the result back asynchronously via the initiator's `nb_transport_bw()` method.


Accepted for publications at DVCON-EUROPE 2025 Design and Verification Conference and Exhibition Europe

### B. Functional Mock-Up Interface

The Functional Mock-Up Interface (FMI) provides a standardized application programming interface specification for the interoperability and co-simulation of dynamic models across heterogeneous simulation environments [12]. FMI facilitates cross-platform model exchange through the encapsulation of computational models FMUs, distributed as compressed archives containing: (1) an XML-based model description file (`modelDescription.xml`), defining the model's interface specification through structured variable declarations (e.g., direction, type, name); (2) platform-specific dynamic link libraries implementing the FMI C-API specification; and (3) optional auxiliary resource files encompassing documentation and model-specific parametric data.

FMI defines three main interactions between FMUs, i.e., Model Exchange (based on numerical integration, handled by a centralized solver), Co-Simulation (where FMUs include their solvers and execute independently), and Scheduled Execution (designed for real-time scenarios). This work focuses on Co-Simulation, that is the most frequent scenario in the context of cyber-physical systems.

FMI 3.0 Co-Simulation specification defines a comprehensive C-API comprising mandatory and optional functions:

- functions to instantiate a FMU, e.g., `fmi3InstantiateCoSimulation`;
- function to run simulation, e.g., `fmi3DoStep`, that allows temporal advancement to run simulation of each FMU;
- getter and setter functions to update FMI variables before and after each `fmi3DoStep`, to allow data exchange between FMUs: `fmi3GetXXX` and `fmi3SetXXX`[1].

### C. Related Work

The Functional Mock-up Interface (FMI) standard has gained substantial traction in both academic research and industrial applications, finding utility across numerous domains [13]. Within the automotive co-simulation sector, several prominent commercial platforms have been developed to leverage FMI capabilities, such as Synopsys Silver [14], Altair Twin [15], Simcenter Amesim [16], and BemNG.tech [17]. While these commercial solutions have achieved significant industrial penetration, they remain proprietary and generally lack native integration capabilities for SystemC models, especially those implementing SystemC TLM.

To address this gap, multiple research efforts have investigated approaches for incorporating SystemC/SystemC TLM components into FMI-based co-simulation frameworks. An initial contribution by [18] presented a methodology for wrapping SystemC models as Functional Mock-up Units (FMUs) using the FMI 2.0 specification, though this approach did not address TLM-specific requirements. Building upon this foundation, later work [19] addressed the integration of TLM-level simulation, focusing on higher abstraction levels compared to RTL. Their proposed methodology begins with a VHDL or Verilog hardware description, which is then transformed into a TLM model of the IP component and subsequently wrapped into an FMU. While the process is automated, it introduces several limitations: the methodology does not directly support SystemC TLM, and an intermediate translation from RTL to TLM is required. OMSimulator [20] represents another FMI-based co-simulation tool, tightly integrated with OpenModelica, an open-source modeling and simulation environment widely adopted in both academia and industry [21]. Through this integration, OMSimulator offers a graphical interface and supports both FMI modalities: model exchange and co-simulation. Although it is build around transaction level co-simulations, it lacks native support for SystemC TLM models.

A few open source efforts are available. [22] introduced VP-Sim, that enables the automated generation of FMI-compliant FMUs through the instantiation of dedicated proxy modules for enabling communication with the system, adding a layer of complexity. Virtual Components Modeling Library (VCML) [23] instead offers a collection of FMU-ready SystemC TLM components for virtual prototyping with FMI support. However, introducing new components requires modifications to the original design, and FMI support is not guaranteed.

To close the gap, this work aims at providing a methodology and an open source tool that automatically wraps SystemC TLM descriptions as FMUs with no modifications of the source code. The open source tool is available at `https://github.com/eml-eda/systemc-fmi`

## III. Methodology

This section presents a comprehensive methodology for encapsulating SystemC TLM designs as FMUs. The approach follows a structured three-phase workflow (outlined in fig. 1):

*A.* Design Selection and Analysis, to collect data structures and identify the communication characteristics;
*B.* FMI Wrapper Generation, to map the TLM payload to FMI variables and implement the required FMI functions;
*C.* Simulation and Validation with an FMI-compliant simulation environment (e.g. FMPy [24]).

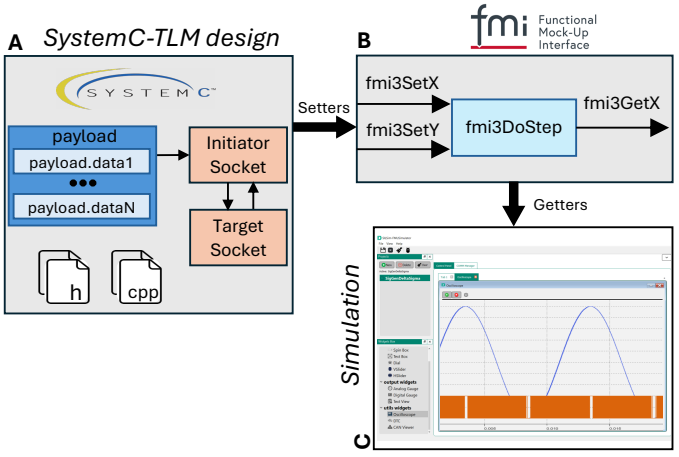


Fig. 1. High-level overview of the proposed workflow

### A. Design Analysis and Simulation Setup

The methodology adopts a non-intrusive approach that preserves the integrity of existing SystemC TLM designs and imposes minimal constraints, ensuring broad applicability across different design styles.

[1] `XXX` stands for the data type, e.g., `fmi3Int8`, `fmi3Boolean`.

*1) Design Prerequisites:* The methodology requires access to the SystemC TLM source code and assumes that the SystemC TLM code is a target module, implementing `b_transport` and `nb_transport_fw` functions. This assumption is reasonable, as TLM modules are typically used to simulate peripherals (e.g., bus interfaces), to be controlled via co-simulation by external modules (e.g., Instruction Set Simulators). Extending the methodology to support an initiator SystemC TLM module would be however straightforward, and will be part of future extensions.

*2) Design Analysis:* The code is parsed to extrapolate the necessary information. The most important element is the data format loaded in the payload as data field. This data is typically organized in a `struct`, that allows to wrap more than one variable. This `struct` is moved to an header file, `payload.h`, so that it can be accessed and imported from multiple files.

```
1  struct payload {
2      sc_dt::sc_int<32> data_in;
3      sc_dt::sc_int<32> data_out;
4  };
```

Listing 1. Structure loaded to payload data and saved in `payload.h`.

Additionally, the SystemC TLM functions implemented in the module are scanned to determine which of the `struct` fields are accessed through read operations (and can thus be considered as inputs received from an initiator) and which are rather set and updated in the function (and are thus outputs generated from the target). In the example reported in listing 1, the `data_in` field is taken in input from the target as data to elaborate, and the resulting output will be uploaded to the `data_out` field of the `struct`. This information will be crucial to build the `ModelDescription.xml` file [12], that contains all necessary metadata for the FMU.

*3) Top-Level and Initiator Module Implementation:* SystemC TLM target modules can not be simulated in isolation, but rather require that their sockets are bound to TLM initiator sockets. For this reason, the methodology implements an *initiator module*, in charge of handling direct communication with the target module, and a top level, that declares, instantiates and connects the initiator and the target.

```
1  void Initiator::send_data()
2  {
3      start_sending.notify(sc_core::SC_ZERO_TIME);
4  }
5  void Initiator::sending_thread()
6  {
7      while(true){
8          wait(start_sending);
9          iostruct data_packet;
10         root_packet.data_in = data_to_send;
11         tlm::tlm_generic_payload payload ;
12         payload.set_command(tlm::TLM_WRITE_COMMAND);
13         payload.set_data_ptr((unsigned char*) &data_packet);
14         ...
15         initiator_socket->b_transport(payload, delay);
16         ...
17     }
18 }
```

Listing 2. Extract of initiator module code.

*4) Initiator module:* The initiator module declares a `tlm_initiator_socket` socket, used to carry as data a `struct`, as defined in the `payload.h` file.

The initiator is in charge of starting communication with the target, via invocation of the SystemC TLM primitives. This is handled with a process `sending_thread` (lines 5-18), that repeatedly sets the payload fields and invokes the blocking or non blocking primitive of the target (an example for blocking communication is reported in listing 2). The process waits on an event `start_sending`, that is fired by a function `send_data`, invoked from outside (as will be explained later, lines 1-4).

*5) Top-level module:* The top-level module is necessary to allow successful SystemC TLM simulation, and listing 3 outlines its implementation. The constructor instantiates initiator and target and performs socket binding (lines 1-6). The top level then includes two main methods. The `set_and_send()` one prepares data to be transferred to the target, copies it to the initiator, and invokes the `send_data` function of the initiator to start communication (lines 7-10). The `retrieve_result()` is vice versa responsible for collecting the output once the transaction has completed (lines 11-13). This solution allows controlling communication with the target and collecting data correctly through methods of the top level, without intruding in the target implementation.

```
1  Top::Top(sc_core::sc_module_name name)
2      : sc_core::sc_module(name) {
3      init = new Initiator("init");
4      root_ = new root("root_");
5      init->initiator_socket.bind(root_->target_socket);
6  }
7  void Top::set_and_send(sc_dt::sc_int<32> data_in_to_send) {
8      init->data_to_send = data_in_to_send;
9      init->send_data();
10 }
11 void Top::retrieve_result(sc_dt::sc_int<32> &result_out) {
12     result_out = init->data_received;
13 }
```

Listing 3. Extract of the top-level module code.

## B. FMI Wrapper Generation

The construction of the FMI wrapper around such system requires generating the FMU interface definition, and implementing the necessary data transfer and primitive mapping to transform FMI APIs into an evolution of the SystemC TLM system.

*1) Model Description Generation:* The first step is the generation of the `modelDescription.xml` file, which defines the FMU's interface. Information collected about the TLM data format is used to populate the file, reporting the name of each `struct` field, annotated with the corresponding direction and type, as depicted in listing 4 for the data structure defined in listing 1. To ensure semantic accuracy, a dedicated mapping (see Table I) aligns each SystemC data type with its corresponding FMI equivalent.

```
1  <fmiModelDescription fmiVersion="3.0" modelName="tlm"/>
2    ...
3    <ModelVariables>
4      <Int32 name="fmi_data_in" valueReference="1"
5           causality="input" start="0"/>
6      <Int32 name="fmi_result" valueReference="2"
7           causality="output"/>
8    </ModelVariables>
9    ...
10 </fmiModelDescription>
```

Listing 4. Extract of the `modelDescription.xml` file generated from the `struct` defined in listing 1

From listing 4, it is possible to appreciate that the `data_in` field is mapped onto a variable `fmi_data_in` (with the `fmi_` prefix) of type `Int32` (corresponding to the initial type `sc_int<32>`). Its direction is considered as input, as it is read as input by the target. Vice versa, the `result` field is tagged as output, as its value is calculated from the target SystemC TLM module, and sent back to the initiator.

TABLE I
DATA TYPE MAPPING BETWEEN SYSTEMC AND FMI

| SystemC Data Type | FMI Data Type |
|---|---|
| `sc_logic` | `fmi3Bool` |
| `sc_bv<N>` | `fmi3Binary` |
| *Signed/Unsigned Integer Types:* | |
| `sc_int<1..8>` / `sc_uint<1..8>` | `fmi3Int8` / `fmi3UInt8` |
| `sc_int<9..16>` / `sc_uint<9..16>` | `fmi3Int16` / `fmi3UInt16` |
| `sc_int<17..32>` / `sc_uint<17..32>` | `fmi3Int32` / `fmi3UInt32` |
| `sc_int<33..64>` / `sc_uint<33..64>` | `fmi3Int64` / `fmi3UInt64` |
| *Floating Point Types:* | |
| `sc_float` | `fmi3Float32` |
| `sc_double` | `fmi3Float64` |

*2) FMI Wrapper Implementation:* The next phase encapsulates the SystemC TLM model using the FMI 3.0 API. The wrapper architecture acts as a bridge between the two domains, coordinating simulation execution and managing data exchange between FMI variables and SystemC sockets. It also ensures correct propagation of inputs and collection of outputs during simulation, while enabling integration into a wider co-simulation context. Central to this mechanism is a structured wrapper (listing 5) that stores:

- a pointer **top* to the top-level SystemC TLM module (listing 3), that allows to access its data structures and to invocate its `set_and_send()` function;
- a variable of type `sc_time` to keep track of simulation time (for the timed versions of TLM);
- the FMI interface variables defined in the XML model description (listing 4, lines 4-5).

```
1  struct WRAPPER_STRUCT {
2     Top *top; // Pointer top-level module
3     sc_time current_time; // Current simulation time
4     fmi3Int32 fmi_data_in; // Input data from FMI domain
5     fmi3Int32 fmi_result; // Output data to FMI domain
6  };
```

Listing 5. SystemC-FMI wrapper struct.

*3) FMI API Implementation:* Conformance with the FMI standard requires implementing the API functions that mediate interactions between the FMI environment and SystemC.

The instantiation and initialization functions (i.e., `fmi3InstantiateCoSimulation` and (`fmi3EnterInitializationMode`) are used to declare and instantiate the top level entity, and to issue a `sc_start(SC_ZERO_TIME)` primitive, that allows construction of the SystemC TLM objects, performs the binding between sockets and initializes the event queue necessary to run the simulation. Vice versa, the `fmi3FreeInstance` function frees memory at the end.

The setter and getter functions (`fmi3SetXXX` and `fmi3GetXXX`, `XXX` being a FMI data type) copy or retrieve the values of the FMI variables, contained in the wrapper, to local variables. The mapping onto SystemC TLM payload data will be explained later on.

The core of the cosimulation is the `fmi3doStep` function, that must advance the simulation of the SystemC TLM subsystem by a certain amount of time (defined by the `CommunicationStepSize` parameter). This requires thus precise synchronization between the SystemC kernel and the FMI runtime, without altering core simulation behavior. An example of implementation is shown in Listing 6. The function accesses the wrapper (defined in listing 5) in line 7 and calculates the requested step size in seconds (line 8). The function then calls the `set_and_send` method of the top-level module to transfer the FMI variables into the TLM payload and to trigger the initiator. It subsequently starts the SystemC TLM simulation for the specified duration using the `sc_start` primitive (lines 11-12). The result calculated by the TLM target is retrieved with the `retrieve_result` function of the top level, and updated to a local variable (lines 13-15). Finally, current FMU time stored in the wrapper is increased to take into account the step size just executed, considering the possibility of anticipated returns due to interrupt-like behaviors or of any error condition detected by the simulation (lines 17-21). This mechanism allows to temporally align the SystemC TLM simulation with any other FMU, and to correctly execute its functionality.

```
1  fmi3DoStep(fmi3Instance instance,
2           fmi3Float64 currentCommunicationPoint,
3           fmi3Float64 communicationStepSize, ...
4           fmi3Boolean* earlyReturn,
5           fmi3Float64* lastSuccessfulTime) {
6
7     WRAPPER_STRUCT* fmu = static_cast<WRAPPER_STRUCT*>(
          ↪ instance);
8     sc_time step_size(communicationStepSize, SC_SEC);
9
10    fmu->top->set_and_send(static_cast<int32_t>(fmu->
          ↪ fmi_data_in));
11    sc_start(step_size);
12
13    int32_t result;
14    fmu->top->retrieve_result(result);
15    fmu->fmi_result = result;
16
17    sc_time next_time;
18    if (!(*earlyReturn))
19      next_time = fmu->current_time + step_size;
20    else next_time = fmu->current_time + (*
          ↪ lastSuccessfulTime);
21    fmu->current_time = next_time;
22
23    return fmi3OK;
24 }
```

Listing 6. `fmi3DoStep` function implementation

## C. FMI-based Simulation

The compilation stage generates a platform-specific executable library (as by the FMI standard): a Dynamic Link Library `.dll` for Windows, a Shared Object Library `.so` for Linux, or a Dynamic Library `.dylib` for macOS. The compilation must ensure that all necessary dependencies are correctly linked and that the final output properly exposes the FMI interface functions required for simulation.

The final phase of the proposed workflow is simulating the generated FMU using suitable simulation tools. A typical simulation flow begins by loading the generated FMU, followed by the configuration of key parameters such as simulation

duration and time step. Initial conditions and input values are then defined before executing the simulation. Once the run is complete, the results can be retrieved and analyzed as needed. For this step, the choice fell on the Python-based FMPy library [24] for its rich feature set, intuitive interface, and ease of integration into automated environments. Although FMPy is used as the default simulation backend in this workflow, the resulting FMUs remain fully compliant with the FMI standard. As such, they can be executed in any FMI-compatible simulation environment, making the solution broadly applicable in both academic and industrial settings.

### D. Automation framework

The whole flow presented in this section has been automated to maximize accessibility and ease of use. To operate, the framework takes as input a SystemC TLM model along with a configuration file (YAML or JSON), that defines the paths to source files, various FMU parameters (e.g., `CommunicationStepSize`), and other necessary metadata for the generation of the `ModelDescription.xml` file.

The automation process is launched via a Python script that handles all subsequent stages, by following the steps detailed in the former subsections. To parse the input files (SystemC TLM design, plus YAML or JSON file) we used regular expressions, and the produced code is written to files. Both Bash scripts and CMake are generated and supported, depending on the user's setup. After successful compilation, the resulting binaries are packaged into a platform-specific FMU (`.dll` for Windows, a `.so` for Linux or a `.dylib` for macOS).

By encapsulating all steps into a single automated pipeline, the framework significantly lowers the barrier for converting SystemC TLM models into FMUs suitable for co-simulation.

## IV. Experimental Results

To assess the effectiveness of our integration methodology, we applied the proposed flow to three case studies. The former two are provided by an industry partner, and are used to evaluate the complexity of the wrapped model, simulation performance of native SystemC TLM simulations (including the target, the initiator, and the top level) against their FMU-based equivalents (executed via FMPy) and the peak memory consumption[2]. The two designs differ in complexity, with one focusing more on communication and the other emphasizing computation. Finally, seamless FMI compatibility is demonstrated through the successful integration of a generated FMU with a Simulink-based FMU.

### A. I2C (Inter Integrated Circuit)

The first case study includes an I2C bus, receiving requests from the FMI interface and handling communication with two slaves: an Arithmetic Logic Unit (ALU) and a Register File. The I2C description comprises a master controller, orchestrating protocol-level operations, e.g., address transmission and acknowledgments, and a slave interface, managing the I2C protocol state machine. The resulting SystemC TLM includes thus 4 modules for a total of 5 processes and 1,072 lines of code. Total payload data includes 7 fields (3 booleans, 2 `uint8`, and 2 enumerative types).

[2]The SystemC TLM runs were measured using a dedicated C++ profiling framework, while FMU co-simulation was monitored in Python using `psutil` [25]. All measurements were averaged over five runs, with simulation lengths varying between 250 and 10,000 `DoStep` invocations.

TABLE II
I2C performance as execution time and memory footprint when increasing the number of `fmi3DoStep` invocations

| **`doStep` (#)** | **250** | | **1,000** | | **10,000** | |
|---|---|---|---|---|---|---|
| Version | TLM | FMU | TLM | FMU | TLM | FMU |
| Time (ms) | 45.44 | 46.50 | 64.93 | 413.00 | 370.94 | 3,750.00 |
| Memory (MB) | 5.40 | 78.40 | 5.28 | 79.27 | 6.30 | 80.85 |

Table II reports the evolution of simulation time and memory footprint when increasing the number of `fmi3doStep` invocations. The overhead ranges from approximately 1.02× with 250 invocations of the `fmi3doStep` function to around 10× with 10,000 invocations. This performance degradation can be attributed to the introduction of additional interfacing layers, resulting in computational and communication overhead, plus to the overhead of time synchronization, data serialization, and inter-process communication. On the contrary, the native SystemC TLM implementation benefits from direct access to efficient time management and transaction-level abstractions, which are partially lost when the module is encapsulated as an FMU.

However, such overhead is reasonable when considering that this allows to cosimulate the wrapped TLM with external tools. As an example, fig. 1.C shows a wrapper SystemC TLM FMU executed and controlled by a Software-in-the-Loop (SIL) environment, used to execute control software interacting with emulated hardware and peripherals, for automotive early virtual prototyping [26].

Memory consumption increases in the FMU-based implementation. For instance, at 250 invocations, the FMU uses 78.4 MB, compared to just 5.4 MB in the SystemC TLM case. This pattern is similar at higher invocation counts, with FMU memory usage being around 80.85 MB at 10,000 invocations, while the native version uses 6.3 MB. The increased memory usage in the FMU version is primarily due to the additional runtime environment and infrastructure required to support the co-simulation interface, and is quite stable when increasing simulation length (with only a 3% increase from 250 to 10,000 `fmi3doStep` invocations).

### B. ECC (Error Correction Code)

The Error Correction Code (ECC) is an crucial component for automotive systems to ensure robust error detection and correction in harsh and electromagnetically noisy environments [27]. The ECC includes XOR-based parity check, dynamic adjustment based on system configuration, and support for both byte mode (8-bit) or word mode (16-bit). The SystemC TLM target includes only 1 module with 3 processes, for overall 1,311 lines of code. Total payload data includes 7 fields (7 logic values and 3 logic vectors).

The ECC module exhibits a similar performance overhead pattern to the I2C module when encapsulated as an FMU (see table III). Execution time degradation ranges from approximately 1.36× for a simulation with 250 invocations of the

TABLE III
ECC PERFORMANCE AS EXECUTION TIME AND MEMORY FOOTPRINT WHEN INCREASING THE NUMBER OF `FMI3DOSTEP` INVOCATIONS

| **doStep (#)** | **250** | | **1,000** | | **10,000** | |
|---|---|---|---|---|---|---|
| Version | TLM | FMU | TLM | FMU | TLM | FMU |
| Time (ms) | 77.72 | 88.30 | 170.11 | 373.90 | 1,185 | 4,945 |
| Memory (MB) | 4.56 | 104.06 | 4.72 | 105.55 | 5.82 | 124.85 |

`fmi3doStep` function to 3.17× for 10,000 invocations. The higher overhead observed at 250 can be attributed to the fixed costs of FMU encapsulation being less effectively amortized over a shorter simulation. In contrast, this overhead becomes proportionally smaller in longer simulations.

Memory usage patterns follow a trend similar to the I2C implementation, with an almost constant overhead of 22×. The higher overhead w.r.t. the I2C design (in avg. 14×) is due to the computation-oriented nature of the ECC design, that thus requires more data transfers and a higher memory occupancy.

### *C. Model-Based Development: SystemC-TLM and Multi-Domain FMU Co-simulation*

The last case study focuses on the interoperability of the generated FMUs with FMUs developed in other modeling environments. As case study, we selected a single-pedal electric vehicle implemented in Simulink (represented in fig. 2). The modeled vehicle features a 1,600 kg mass, 90 horsepower drivetrain, and a unified pedal interface that provides both acceleration and regenerative braking functionality. The exported Simulink FMU encapsulates the complete vehicle dynamics through 19 variables: time, input torque request (Nm), output vehicle speed (km/h), and 16 configurable parameters defining the vehicle characteristics. The Simulink design was exported as an FMU using the CATIA FMI-Kit [28]. To complement this plant model, a SystemC TLM Electronic Control Unit (ECU) was developed to generate representative driving scenarios. The ECU outputs the commanded torque values to be fed to the vehicle plant.

The SystemC TLM design was packaged as an FMU with the approach proposed in this work. The co-simulation was executed using the FMPy library to orchestrate the interaction between both FMUs.

Simulation results (reported in fig. 3) demonstrate successful coupling between the SystemC TLM ECU (top) and the Simulink vehicle model (bottom) over a 35-second test scenario. The torque request profile generated by the ECU exhibits a characteristic driving cycle, beginning with gradual acceleration (0-120 Nm), maintaining steady-state operation, followed by regenerative braking with negative torque values (down to -80 Nm), and concluding with a return to cruise conditions. Vehicle speed response shows appropriate dynamic behavior, accelerating from rest to approximately 120 km/h during positive torque phases and decelerating during regenerative braking periods. The response exhibits realistic vehicle inertia characteristics, with smooth speed transitions that follow the torque command profile with expected delays inherent to vehicle dynamics.

The piecewise constant curve of the Simulink output (bottom) is due to the different time steps used by the two cosimulations (0.1s for SystemC TLM, 1s for Simulink). This

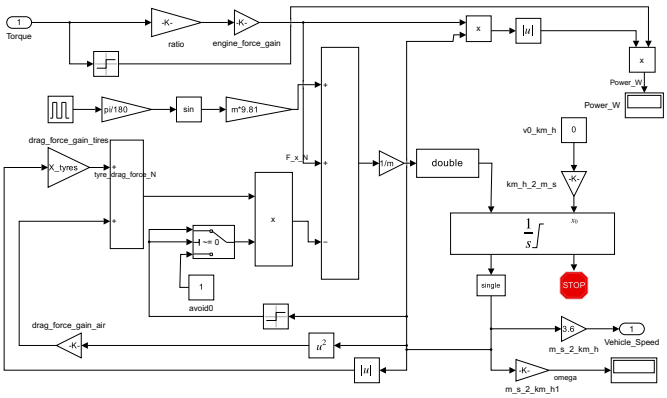


Fig. 2. Simulink model of a single-pedal electric vehicle

confirms that the time synchronization mechanism is robust, as it allows correct cosimulation with FMUs running with a different `CommunicationStepSize` setting.

This successful co-simulation confirms that the FMUs generated by our framework are fully compliant with the FMI standard and are designed to be modular and interoperable. As such, they can be seamlessly integrated with FMUs developed using other tools and modeling environments, enabling flexible and multi-domain simulation workflows.

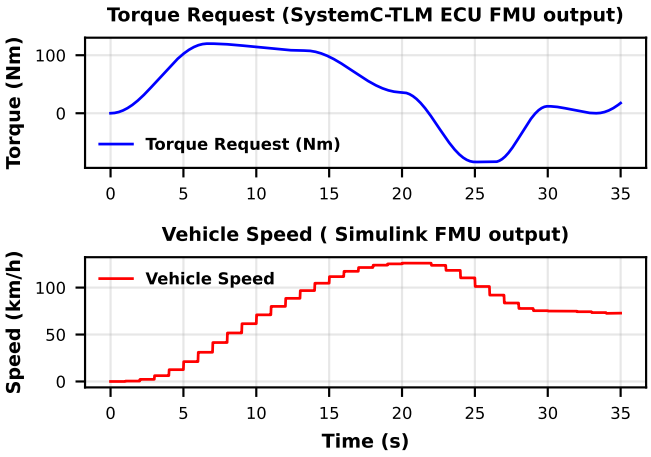


Fig. 3. Cosimulation of the SystemC TLM ECU with the Simulink model in fig. 2: torque request (top) and vehicle speed (bottom).

## V. CONCLUSIONS

In this paper, we presented an open-source automated framework for integrating SystemC TLM models with the FMI standard, facilitating co-simulation and cross-domain interoperability. The framework and the accompanying open source tool automatically generate FMUs from unmodified SystemC TLM models, ensuring tool-independent and non-intrusive integration. Experimental results demonstrate its ability to handle a wide range of designs efficiently, with consistent memory usage and moderate performance overhead. Additionally, the experimental results prove the straightforward integration with other FMUs. Future work will focus on extending support for additional SystemC TLM features and integrating with Instruction Set Simulator (ISS) based environments.

## REFERENCES


[1] Z. Zhang *et al.*, "Co-simulation framework for design of time-triggered cyber physical systems," *Simulation Modelling Practice and Theory*, vol. 43, p. 16–33, 04 2014.

[2] G. Martin, "Embedded system design: Modeling, synthesis, and verification," *Design & Test of Computers, IEEE*, vol. 27, pp. 82 – 83, 05 2010.

[3] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.

[4] A. Mahmoudi *et al.*, "A systematic mapping study on SystemC/TLM modeling capabilities in new research domains," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 4, Jun. 2025.

[5] P. A. Fritzson, *Principles of object-oriented modeling and simulation with modelica 3.3*, 2nd ed. Nashville, TN: John Wiley & Sons, Nov. 2014.

[6] E. Widl *et al.*, "FMI-based co-simulation of hybrid closed-loop control system models," in *Proc. of ICCSE*, 2015, pp. 1–6.

[7] A. Haider *et al.*, "Modeling and simulation of automotive FMCW RADAR sensor for environmental perception," *IEEE Open Journal of Intelligent Transportation Systems*, vol. 6, pp. 433–455, 2025.

[8] F. Perabo and M. Zadeh, "Multiphysics modeling and co-simulation of ship electric power and propulsion systems for virtual testing and verification," *IEEE Transactions on Transportation Electrification*, vol. 11, no. 1, pp. 5108–5121, 2025.

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

[10] "accellera.org," https://www.accellera.org, July 2009.

[11] T. Grötker *et al.*, *System Design with SystemC*. Springer, 01 2002.

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

[13] S. Hansen *et al.*, "The FMI 3.0 standard interface for clocked and scheduled simulations," *Electronics*, vol. 11, p. 3635, 11 2022.

[14] Synopsys, "Silver: Software in the loop for virtual ecus — synopsys," https://www.synopsys.com/verification/virtual-prototyping/silver.html#features, 2025.

[15] Altair, "Altair twin," https://altair.com/twin-activate, 2025.

[16] Siemens, "Simcenter amesim," https://plm.sw.siemens.com/en-US/simcenter/systems-simulation/amesim/, 2025.

[17] BeamNG, "Beamng.tech," https://beamng.tech/, 2025.

[18] S. Centomo *et al.*, "Using SystemC cyber models in an FMI co-simulation environment: Results and proposed FMI enhancements," in *Proc. of Euromicro DSD*, 2016, pp. 318–325.

[19] ——, "Transaction-level functional mockup units for cyber-physical virtual platforms," in *Proc. of FDL*, 2018, pp. 5–8.

[20] L. Ochel *et al.*, "OMSimulator - Integrated FMI and TLM-based Co-simulation with Composite Model Editing and SSP," in *International Modelica Conference*, 02 2019, pp. 69–78.

[21] Open Source Modelica Consortium, "Openmodelica," https://openmodelica.org/.

[22] S. E. Saidi *et al.*, "Fast virtual prototyping of cyber-physical systems using SystemC and FMI: ADAS use case," 10 2019, pp. 43–49.

[23] "VCML Virtual Platform: Open-Source SystemC TLM-2.0 Library - machine-ware.de — machineware.de," https://www.machineware.de/products/vcml-virtual-platform, 2025.

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

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

[26] D. Group, "Silsim," https://www.dumarey.com/solution/virtual-prototyping/, 2025.

[27] A. Tadimarri *et al.*, "Driving towards safety: The role of ECUs and IMUs in advanced driver-assistance systems (ADAS)," *International Journal For Multidisciplinary Research*, vol. 6, 04 2024.

[28] T. Sommer *et al.*, "Catia-Systems/FMIKit-Simulink," https://github.com/CATIA-Systems/FMIKit-Simulink, 2024.