FMI Meets SystemC: A Framework for Cross-Tool Virtual Prototyping

Nils Bosbach[1] Meik Schmidt[1] Lukas Jünger[2] Matthias Berthold[3] Rainer Leupers[1]

[1]RWTH Aachen University, Aachen, Germany, {bosbach,schmidtm,leupers}@ice.rwth-aachen.de
[2]MachineWare GmbH, Aachen, Germany, lukas@mwa.re
[3]tracetronic GmbH, Dresden, Germany, Matthias.Berthold@tracetronic.de

Abstract

As systems become more complex, the demand for thorough testing and virtual prototyping grows. To simulate whole systems, multiple tools are usually needed to cover different parts. These parts include the hardware of a system and the environment with which the system interacts. The Functional Mock-up Interface (FMI) standard for co-simulation can be used to connect these tools.

The control part of modern systems is usually a computing unit, such as a System-on-a-Chip (SoC) or Microcontroller Unit (MCU), which executes software from a connected memory and interacts with peripherals. To develop software without requiring access to physical hardware, full-system simulators, the so-called Virtual Platforms (VPs), are commonly used. The IEEE-standardized framework for VP development is SystemC TLM. SystemC provides interfaces and concepts that enable modular design and model exchange. However, SystemC lacks native FMI support, which limits the integration into broader co-simulation environments.

This paper presents a novel framework to control and interact with SystemC-based VPs using the FMI. We present a case study showing how a simulated temperature sensor in a SystemC simulation can obtain temperature values from an external tool via FMI. This approach allows the unmodified target software to run on the VP and receive realistic environmental input data such as temperature, velocity, or acceleration values from other tools. Thus, extensive software testing and verification is enabled. By having tests ready and the software pre-tested using a VP once the physical hardware is available, certifications like ISO 26262 can be done earlier.

Keywords: SystemC, TLM, FMI, Virtual Platform

1 Introduction

In today's rapidly evolving technology landscape, systems are becoming more complex, requiring advanced development cycles and rigorous testing methodologies. As industries, such as automotive, tend to solve more problems in software rather than hardware, early and intensive software testing is paramount (Ondrej Burkacky et al. 2018). To start software development and testing while the hardware is still under design, Virtual Platforms (VPs), also known as *Level 4 Virtual Electronic Control Units (L4 vECUs)*, can be used. It has been shown that the earlier bugs are found, the lower the cost of rectifying them. Fixing a bug during the development phase can be up to 100 times cheaper than fixing it while the product is already in use (Boehm 1976). VPs are the key enabler of early software testing. By simulating a System-on-a-Chip (SoC), the unmodified target[1] software can be developed and tested without requiring access to the physical hardware.

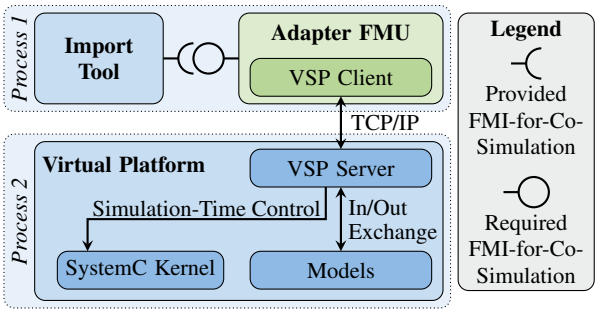


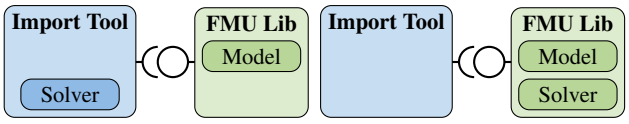
Figure 1. SystemC-FMU adapter approach.

The IEEE-standardized framework for VP design is SystemC with its Transaction-Level Modeling (TLM) extension (IEEE Standards Association and others 2023). While VPs are well-suited tools for software development and verification, they reach their limits when it comes to realistic testing of different scenarios. Because embedded systems, such as Electronic Control Units (ECUs), interact with their physical environment by reading sensor values and controlling devices, it is not enough to simulate only the system itself. In addition, environmental effects must also be considered and simulated.

To simulate physical systems and define complex test cases, there are many tools available. In this paper, we show how a SystemC-based simulation can be connected to those tools using the Functional Mock-up Interface (FMI) standard (Modelica Association 2024). The FMI co-simulation standard allows to synchronize the simulation time of multiple simulations and to exchange data.

Figure 1 shows the integration of our approach in a co-simulation. We developed an Functional Mock-up Unit (FMU) that can be imported by any FMI-compatible tool to control a VP. This FMU exposes the specified output of

[1]*target*: architecture that is simulated; *host*: architecture that runs the simulation



(a) Model Exchange. **(b)** Co-Simulation.

Figure 2. FMI types.

VP and accepts the inputs that are forwarded to VP. The VP runs in a second process, either on the same host as the FMI import tool, or on a different one. The connection between the FMU and the VP is TCP/IP based.

We present a case study where we show how an open-source VP can be connected to tracetronic's *ecu.test* tool using FMI. ecu.test allows defining complex test scenario where environmental data can be specified and responses of the tested system are compared against expectations.

In the end, we propose a software design flow that uses our integration for early testing and verification. This design flow helps to shorten the time to reach specifications like ISO 26262 (ISO/TC 22/SC 32 2018).

2 Background and Related Work

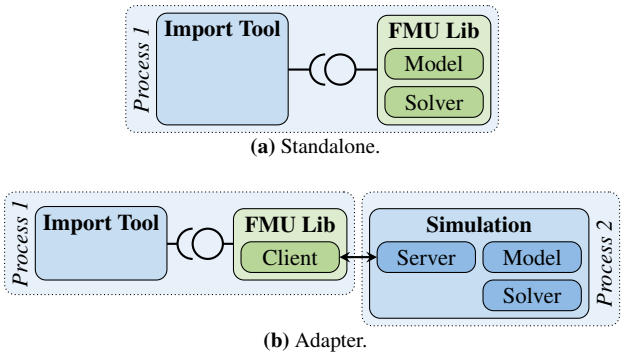
This section provides an overview of the related work that forms the foundation of this study. Section 2.1 explains the FMI standard and the features used in this work. Section 2.2 provides an overview of virtual prototyping, including background information on the industry-standard framework SystemC and the Virtual Components Modeling Library (VCML), which builds on top of SystemC. In Section 2.3, an overview of existing approaches to integrate SystemC modules into an FMI co-simulation is given.

2.1 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) is an open standard designed to enable seamless integration and interoperability of simulation models from various tools and vendors (Modelica Association and MODELISAR Consortium 2024). Initially introduced by the Modelica Association in 2010, FMI has become a widely adopted standard for modular and tool-agnostic simulation workflows and is supported by more than 200 tools.

Simulation models are encapsulated into so-called FMUs. This encapsulation process is performed by *export tools*, which are typically provided by simulation-software vendors. FMUs can then be imported and utilized by *import tools*, which coordinate the simulation and facilitate data exchange between multiple FMUs.

The original standard supports two interface types, *Model Exchange* (Modelica Association 2010b) and *Co-Simulation* (Modelica Association 2010a), as illustrated in Figure 2. As physical systems are often described by differential equations, a numerical solver is needed to solve and evaluate these equations. The difference between the FMU interface types is primarily in the placement of the numerical solver. In the Model-Exchange use-case, the solver is not part of the FMU and therefore needs to be implemented in the import tool.



(a) Standalone.

(b) Adapter.

Figure 3. FMI Co-Simulation scenarios.

With the introduction of FMI 3.0 (Modelica Association and MODELISAR Consortium 2024), a third interface type, *Scheduled Execution*, was added. For this work, we only use the Co-Simulation interface, so the other ones are not further discussed. We demonstrate how a full VP can be encapsulated into an FMU. Because VPs usually operate at a much higher level of abstraction than differential equations, we do not require a numerical solver. Consequently, Co-Simulation is the preferred approach.

When the Co-Simulation interface is used, two scenarios are possible, a *standalone* and an *adapter* scenario (Blochwitz et al. 2011). Both scenarios are visualized in Figure 3. The FMU contains a shared library (*FMU Lib*) that is loaded by the import tool. It implements standardized FMI functions, the import tool calls. In the standalone scenario, as shown in Figure 3a, the shared library contains the full simulation model. The simulation model can be directly executed in the process of the import tool.

In the adapter scenario, illustrated in Figure 3b, the shared library acts as an interface to another process that executes the simulation. This connection can, e.g., be established using network-based protocols, such as TCP/IP. One key advantage of this approach is the ability to run the import tool and simulation on different machines, which can offer potential performance benefits by distributing computational load. Additionally, this configuration allows the simulation and import tool to run on different Operating Systems (OSs) or even different hardware architectures, such as x86 and Arm.

2.2 Virtual Prototyping

Virtual prototyping is a technique in which electronic systems are modeled entirely in software. This work focuses on VPs, virtual prototypes designed for software development for the simulated target platform. VPs can simulate the execution of an unmodified target-software stack. In the automotive domain, they are referred to as *L4 vECUs*.

The industry-standard framework for VP development is SystemC with its TLM extension (IEEE Standards Association and others 2023). SystemC standardizes the interface and functionality of a C++ library that features interfaces, data types, and a Discrete Event Simulation

DOI
10.3384/ecp218545

```
<fmiModelDescription ... fmiVersion="3.0" modelName="myVP" ...>
    <CoSimulation modelIdentifier="myVP" needsExecutionTool="true"
    canHandleVariableCommunicationStepSize="true" ... />
    <DefaultExperiment startTime="3" stopTime="5" stepSize="0.01"/>
    <ModelVariables>
        <Float64 name="time" valueReference="0" causality="independent"
    variability="continuous" />
        <Float32 name="system.max31855.temp" valueReference="1" causality="input"
    variability="continuous" start="10.0" />
        <UInt32 name="system.gpio.data" valueReference="2" causality="output"
    variability="discrete" />
    </ModelVariables>
    <ModelStructure>
        <InitialUnknown valueReference="1"/>
        <Output valueReference="2"/>
    </ModelStructure>
     <Annotations>
        <Annotation type="VCML">
            <VP host="localhost" port="8888"  executable="resources/vp" ... />
        </Annotation>
      </Annotations>
</fmiModelDescription>
```

Listing 1. Model-description file.

(DES)-based scheduler to simulate parallelism. A simulation usually contains multiple modules that can be connected via TLM sockets. TLM abstracts the underlying communication protocol and offers low-overhead data exchange. The standardized interfaces guarantee seamless integration of different models into a simulation.

On top of SystemC, modeling libraries provide additional functionalities. One open-source modeling library is VCML (MachineWare 2024). VCML extends SystemC with essential components, such as register models, prebuilt hardware components, and specialized TLM-based communication protocols. A key feature used in this work is the VCML Session Protocol (VSP). VSP is a remote-control protocol that is based on GDB's Remote Serial Protocol (RSP) protocol (Free Software Foundation, Inc. 2024). VCML includes a VSP server module, which allows clients to connect via TCP to control the simulation.

Another crucial VCML feature used in this work is *properties*, which encapsulate configuration variables within modules. These properties can be set at VP launch and accessed via the VSP protocol, facilitating dynamic reconfiguration.

2.3 Related Work

Previously, various publications have described methods for integrating SystemC-based simulations into FMI co-simulations. Most of these approaches are based on the FMI 2 standard and involve tight coupling between the FMI interface and the VP.

One approach integrates an FMI import tool directly into the VP to co-simulate SystemC models with FMUs that provide environmental models (Safar et al. 2018). This approach means that the VP cannot be used with other import tools, e.g., for setting up test scenarios, as described in Section 4. Furthermore, the VP is no longer standalone, so co-simulated FMUs are always required.

For detailed Register-Transfer Level (RTL)-like SystemC simulations, a different approach can be used to expose SystemC ports to FMI (Centomo, Deantoni, and De Simone 2016). This approach also requires modifications to the VP, which prevents a standalone simulation. In our work, we focus on SystemC-TLM-based VPs that use a higher abstraction level to achieve increased performance.

For SystemC Analog/Mixed-Signal (AMS) simulations, wrapper classes can be created to wrap models and connect the needed inputs and outputs to an FMI integration (Krammer et al. 2015). This kind of simulation also targets more detailed abstraction levels and requires modifications to the simulation.

In another approach, *FMI variables* are introduced, that can be bound to different protocols like CAN, or I^2C (Saidi et al. 2019). When a variable changes, different actions can be carried out. For example, a CAN frame carrying the updated variable as payload can be injected onto the bus. This approach is useful for simple sensors that use common protocols. An advantage is that the sensor itself does not need to be part of the VP but can be completely used from the FMI integration. This approach also does not allow for standalone use of the VP.

In our work, we present an approach that does not require VP modifications and therefore no source-code access. The requirement is that theVP is based on the used modeling library VCML and the VSP server is not disabled. The VP can then be used without the FMI integration when environmental input are not needed. When environmental inputs become important, the VP can be integrated into an FMI co-simulation without any changes by using our standalone FMI-adapter FMU.

3 Approach

In this section, we present our approach. First, Section 3.1 describes the application scenario. Then, Section 3.2 explains how we integrate a VCML-based VP into an FMU. In the end, Section 3.3 provides further details on the FMU

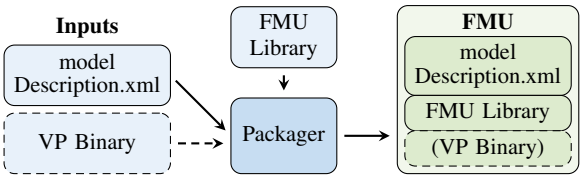


Figure 4. FMU creation.

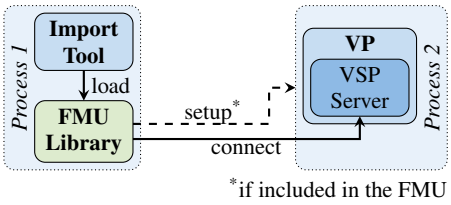


Figure 5. Working principle.

implementation and the integration process.

3.1 Application Scenario

Imagine a VP that models an ECU that consists of various peripherals including sensing devices for measuring environmental conditions. Such a VP is crucial for target software development and verification but traditionally lacks dynamic environmental modeling capabilities. For instance, a virtual model of a temperature sensor simulates all hardware interfaces other components can access but typically returns constant values during standalone VP execution, limiting its ability to test software responses to varying conditions. Directly integrating environmental simulations, such as temperature models, into device models like sensors would have multiple drawbacks:

- **Simplicity**: The model should be kept simple to have a high performance. Complex algorithms to model the environment would unnecessarily slow down the model and increase its complexity when the environmental simulation is not needed.
- **Genericity**: The model should only reflect the behavior of the hardware to use it in various scenarios. An environmental simulation model heavily depends on the scenario and not the component itself.
- **Specialization of Tools**: There are well-established tools to create environmental models, e.g., based on physical equations or test cases.

Because of those reasons, the returned temperature value from our model is read from a VCML property (see Section 2.2). This means that in a standalone simulation, the temperature value is set at startup and remains constant throughout execution. While this setup allows for full software stack execution including sensor interaction, it does not support testing of how the software reacts to changing environmental conditions.

To overcome this limitation, our FMI integration enables co-simulation of the VP together with external tools that dynamically provide environmental data. This approach allows for realistic scenario testing and a more comprehensive evaluation of the system's behavior using dedicated tools to solve various tasks.

3.2 FMU Creation

To create an FMU from a VCML-based VP, our *Packager* tool can be used. Figure 4 illustrates the creation process. The resulting FMU includes our VP-independent FMU library and a model-description file, which defines its properties. Section 3.3 explains the working principle of the FMU library in detail. A simplified example of a model-description file is shown in Listing 1.

The `ModelVariables` element lists the VP's inputs and outputs that are accessible via FMI. Each variable's name corresponds to the property name within the module hierarchy of the VP. For example, the `system.max31855.temp` property defines the property `temp` of the SPI temperature sensor `max31855`, which is a submodule of the `system` module. It is sufficient to list the hierarchical name of the property in the model-description file. There are no changes in the VP needed to expose the property. The VSP server can clearly identify and access the properties by their hierarchical names. Additionally, data types are specified, and it is indicated whether the property functions as an input or output. For inputs, a start value can be assigned. During simulation, FMI uses the `valueReference` numbers when updating the variables.

The `Annotations` element defines VP-specific properties using the available attributes listed in Table 1. The VP binary can be embedded within the FMU and launched by the FMU. In this case, the `executable` attribute points to the relative path of the VP binary within the FMU (e.g., `resources/my_vp`). Additionally, the `host` is set to *localhost*, and the `port` specifies the TCP port the VP's VSP server is listening.

If the VP is not embedded in the FMU, e.g., because it is manually launched on a separate machine, only the `host` and `port` attributes need to be configured. The `host` attribute defines the hostname of the machine that executes the VP. This machine needs to be reachable by the import tool via a TCP connection.

3.3 Working Principle

An exported FMU can be loaded by an FMI-3-compatible import tool. Figure 5 illustrates the workflow.

Table 1. Attributes of the VCML annotation node.

Attribute	Meaning
`executable`	Path of the VP executable
`args`	Args that are passed to the VP executable during startup
`host`	IP address or hostname of the machine that executes the VP
`port`	Listening port of the VP server

DOI
10.3384/ecp218545

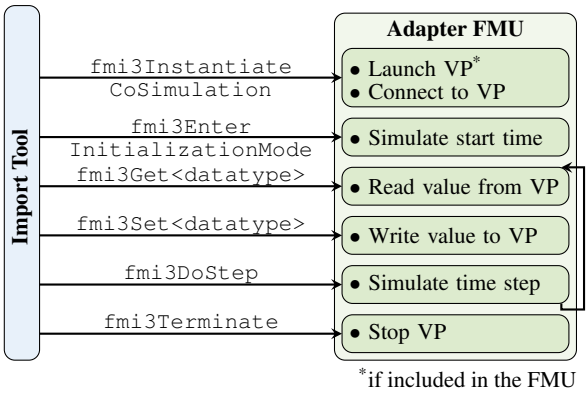


Figure 6. FMI functions implemented by our FMU.

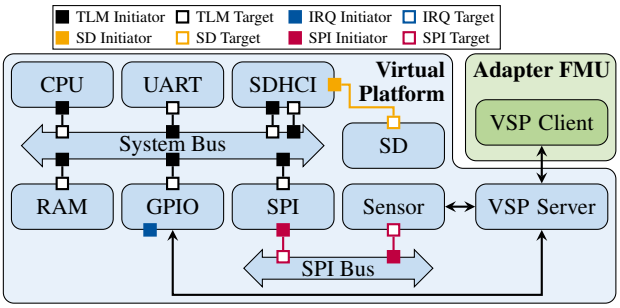


Figure 7. AVP64-based ECU with connected FMI adapter.

The import tool unpacks the FMU and reads the model-description file. It then loads the FMU library (*.dll* or *.so*). Afterward, the import tool can call the standardized FMI functions that are implemented by the FMU library. If the VP binary is packed into the FMU, the FMU library starts the VP in a second process. The library establishes a TCP-based connection to the VSP server of the VP and uses this connection to control the simulation.

Figure 6 shows the FMI functions our FMU implementation. The tasks that are executed by those functions are explained in the following in more detail.

First, the `fmi3InstantiateCoSimulation` function is usually called. If the VP is part of the FMU, a second process is started to run the VP. Parameters can be passed to the VP via command-line arguments. If the VP is not part of the FMU, it needs to be started manually. This can be done either on the same machine or on a different one that is located in the same network as the machine that runs the FMU. In this case, the corresponding hostname of the machine that runs the VP needs to be set in the `VCML` node of the model description's `Attribute` node. Once the VP is started, the VSP server listens on the TCP port for a connection and suspends the VP until the FMU is connected. The FMU connects the VSP server of the VP.

When the `fmi3EnterInitializationMode` function is called by the import tool, a command is sent to the VSP server to simulate until the virtual simulation time of the SystemC simulation reaches the specified `startTime`. The start time parameter of the FMU can be used to, e.g., skip the boot process of the OS running on the system. If the start time is set to 0, our FMU does nothing in this step. After the `fmi3EnterInitializationMode` function call, the VP is ready for simulation.

The import tool can read model variables that have been declared as *output* in the model description by calling the `fmi3Get<datatype>` functions. For example, `fmi3GetFloat64` is used when the variable is declared as *Float64* in the model description. The variable is referenced by its `valueReference` property. The FMU translates the `valueReference` into the `name` using the model description. This name corresponds to the hierarchical name of the property in the VP (see Section 3.2). This hierarchical name can be directly used by the VSP server to identify, read, and write the property value within the simulation. All available properties of the simulation can be used as FMU inputs and outputs by simply specifying their hierarchical name in model description.

To update an FMU input, the `fmi3Set<datatype>` function can be used in the same way. Again, the variables are referenced by their `valueReference` property. The updated input values are directly sent to the VP to update the corresponding model properties.

After variables have been updated, the `fmi3DoStep` function can be called to simulate a time step. The amount of simulation time that is simulated on a call is usually defined by the `stepSize` attribute in the model description (see Listing 1). Our FMU sends a command to the connected VP to simulate the step. The VP's SystemC scheduler simulates events until the virtual simulation time is increased by the specified step. This allows the virtual CPU model to execute target software, and interact with other simulated peripherals. The import tool repeatedly calls the `fmi3Get<datatype>`, `fmi3Set<datatype>`, and `fmi3DoStep` functions as long as the test or the simulation is running.

To end the simulation, the `fmi3Terminate` function is called, which stops the VP and shuts down the simulation.

4 Case Study

To demonstrate the effectiveness of our FMI integration approach for SystemC-based VPs, we present a case study of a software-based Schmitt trigger application with a temperature input. We show how the different components of the system can be efficiently modeled and integrated into an FMI-based co-simulation using our developed framework. The full system consists of three parts:

ECU: The ECU consists of an ARMv8 processor, memory, and peripherals. The components and their connections are shown in Figure 7. A *MAX31855* (Maxim Integrated 2015) temperature sensor (denoted as *Sensor* in Figure 7) can measure the ambient temperature. The sensor is connected via an SPI interface to the SPI controller, which is accessible by the CPU. Additionally, the system features a General Purpose Input/Output (GPIO) controller that allows the CPU to control external signals. To simulate the ECU hardware, the open-source, SystemC-TLM-based ARMv8 Virtual Platform (AVP64) (Jünger et

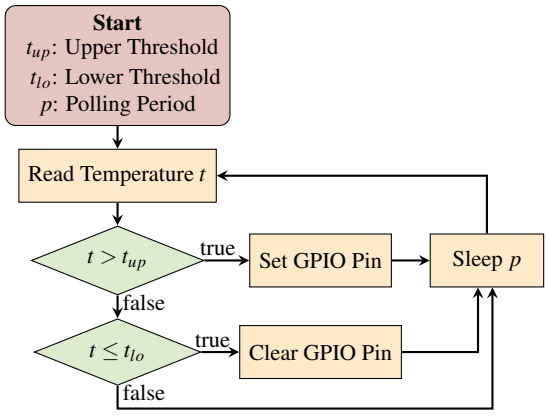


Figure 8. Schmitt-trigger algorithm executed by the VP.

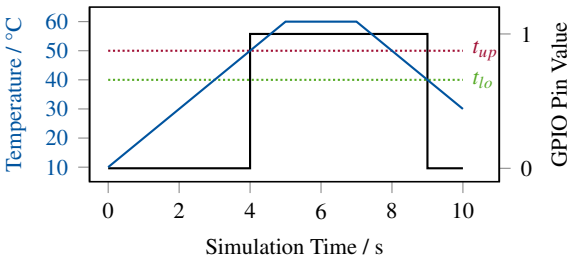


Figure 9. Temperature input and expected GPIO-pin output for $t_{lo} = 40\,°\text{C}$ and $t_{up} = 50\,°\text{C}$.

al. 2019; Jünger 2023) is used. The VP is capable of executing the full target-software stack.

Software Application: The ECU boots a *Buildroot*-based Linux OS (Buildroot 2024). Upon completion of the boot process, a simple Schmitt-trigger application is automatically launched. The Schmitt-trigger application implements the algorithm depicted in Figure 8. This algorithm monitors the temperature input of the sensor and sets or clears a GPIO pin based on the predefined threshold values t_{lo} and t_{up}. When the temperature exceeds t_{up}, the GPIO pin is set. When it falls below t_{lo}, the GPIO pin is cleared. The application periodically polls the temperature sensor at a configurable interval p. An example of a temperature curve and the expected output for $t_{lo} = 40\,°\text{C}$ and $t_{up} = 50\,°\text{C}$ is shown in Figure 9.

Environmental Impacts: The ECU interacts with its environment by measuring the ambient temperature and reacting to changes. To add the simulation of changing environmental conditions and enable comprehensive testing with dynamic environmental scenarios, we package the VP into an FMU using our FMI integration approach.

This allows for co-simulation with external tools that can provide varying temperature inputs. The FMU configuration for our scenario includes one input and one output as shown in Figure 10. The input is connected to the temperature property of the MAX31855 sensor model, and the output retrieves the GPIO pin value from the GPIO controller (see Figure 7). There are no changes in the VP required to be connected to the FMU. It is sufficient to specify the hierarchical names of the properties for the temperature value and the GPIO pins in the model description (see Section 3.2).

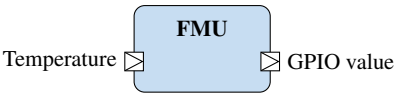


Figure 10. FMU with inputs and outputs.

4.1 Target-Software Verification

Software verification and testing play a crucial role during the design process of a system. The earlier a bug is found, the less effort and cost it will take to fix it. As described in Section 1, if a bug is first found during product operation, the cost of fixing it can be up to 100 times higher than if it is found during software development (Boehm 1976). By integrating the VP into an FMI co-simulation using our approach, complex scenarios can be tested to find bugs as early as possible. This leads to lower costs, a shorter time to market, and overall improved development efficiency.

A typical measure that is used to determine the percentage of tested code is code coverage (Ivanković et al. 2019). Traditional line coverage (Miller and Maloney 1963) describes the number of executed lines of code divided by the total number of lines of code. More advanced metrics like Modified Condition/Decision Coverage (MC/DC) (Kelly J. Hayhurst et al. 2001) are required for ISO-26262-based Automotive Safety Integrity Level (ASIL) D certification (ISO/TC 22/SC 32 2018). Besides counting the executed lines of code, MC/DC adds additional metrics like counting if Boolean expressions have been evaluated as both, true and false. The VP can directly extract coverage metrics of the target software after execution without requiring instrumentation of the target software (Bosbach et al. 2024).

For extensive testing and validation, we utilize *ecu.test* from *tracetronic GmbH* (tracetronic GmbH 2024). This tool allows defining complex test scenarios, specify environmental input data, and compare the system's responses against expected outcomes to test if the system fulfills its requirements. ecu.test is commonly used for ECU testing and verification. It has native support for FMI.

For easy setup and test configuration, ecu.test can be executed on Windows-based machines and controlled using a Graphical User Interface (GUI). The VP is executed on a Linux cloud server. ecu.test loads our adapter FMU on the Windows machine. The FMU connects via a TCP/IP connection to the VSP server that runs on a Linux machine in the same network. For scaled testing, the whole setup can be executed on a single Linux machine using tracetronic's Linux version of ecu.test. By combining our VP FMU with ecu.test, we can perform realistic and thorough testing of the Schmitt trigger application by simulating various temperature conditions. An example of a successful test run is illustrated in Figure 11.

Execution traces of the CPU model can be used to generate coverage reports. To showcase the usefulness of a coverage report, we turned on the code-coverage feature

DOI
10.3384/ecp218545

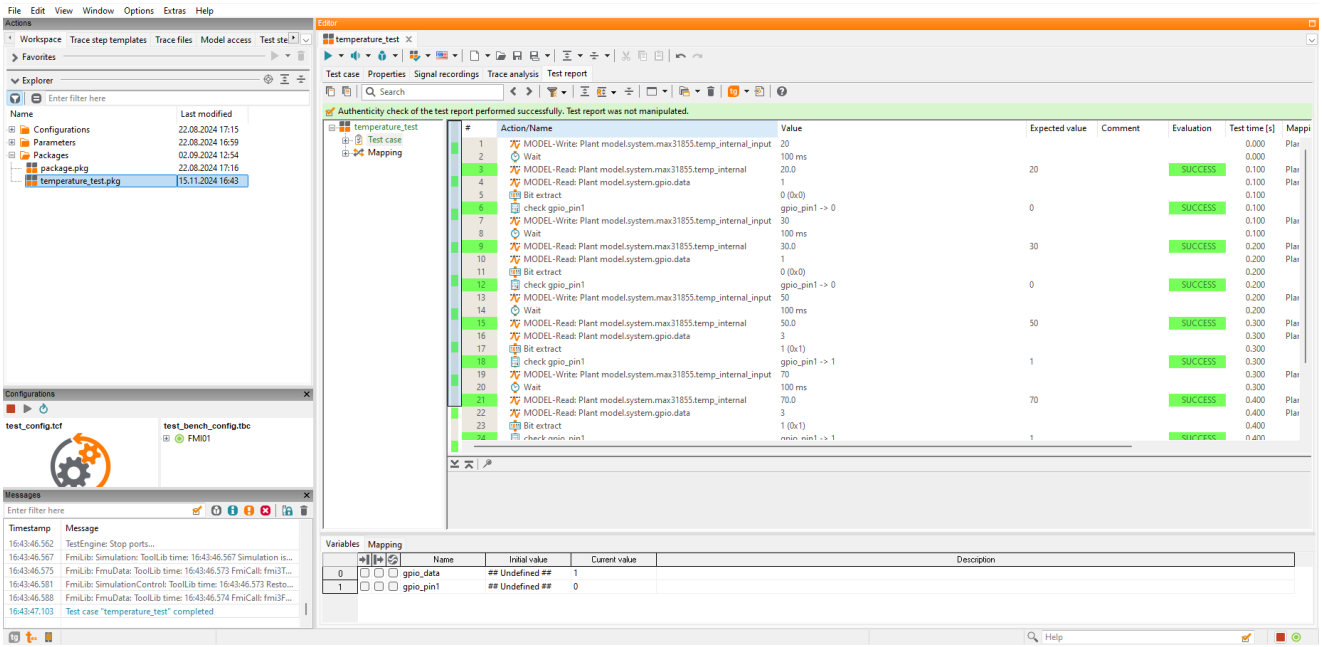
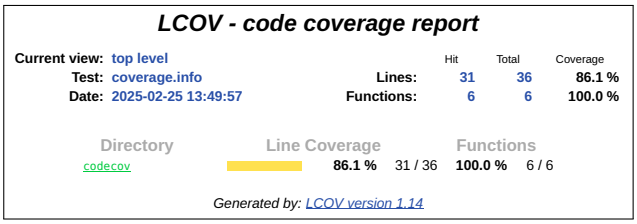
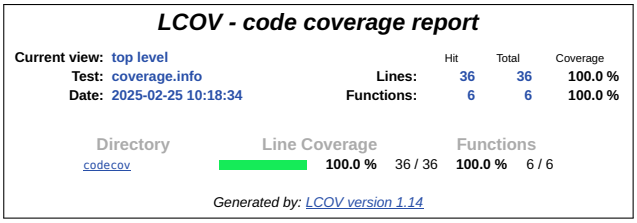


Figure 11. Successful test run in ecu.test.



(a) Constant temperature (standalone VP).



(b) Temperature controlled by ecu.test.

Figure 12. Coverage results for different configurations.

of the VP to generate execution traces and converted them to a *lcov*-based coverage report (Linux Test Project 2023).

Figure 12 visualizes coverage results for the presented Schmitt-trigger application. Figure 12a shows the report for the standalone VP without the FMI integration and ecu.test. Since the temperature value is constant in this case, the algorithm never executes the path that sets the GPIO pin (see Figure 8). This leads to a line coverage of 86.1 %. When ecu.test is connected and test scenarios are executed, the line coverage is increased to 100 %.

This simple example shows that the consideration of environmental influences is necessary for extensive testing. The effectiveness of the applied testing is directly reflected by the increased code coverage.

A more realistic industrial-grade application is, e.g., the simulation of an engine control unit in a VP. The various sensor inputs (e.g., position, temperature, and veclocity sensors) can be read from FMI. Output-signal characteristics, such as the duty cycle and frequency of Pulse Width Modulation (PWM) signals to control actuators like fuel injectors, ignition coils, and idle speed control valves, are forwarded to the importing tool for further processing.

5 Design-Flow Integration

Based on our integration, we propose a design flow that can be used for early target software verification and creation of tests. An overview is shown in Figure 13. At the beginning of the design process, a VP can be created according to the specification of the system. The target software can be developed and tested using this VP. Once the target software development is done, our FMI integration can be used to integrate the VP into a co-simulation and develop test scenarios. Edge-cases can be tested to find bugs in the target software early. The extracted coverage metrics guide verification and test engineers to cover all parts of the target software.

To fulfill standards like ISO 26262 (ISO/TC 22/SC 32 2018), the satisfaction of the requirements needs to be tested. Our FMU can be used to design those tests using a VP. Once physical hardware is available, the hardware can directly be tested using the already developed tests.

In contrast to physical hardware, the virtual model can be scaled arbitrarily, distributed around the world within seconds, and easily integrated into Continuous Integration/Continuous Delivery (CI/CD) flows. CI/CD allows for continuous and automatic testing during the development process to reduce costs by finding bugs earlier.

Once the physical hardware is available, the same test can be executed on the real hardware by replacing the adapter FMU by an hardware-communication interface.

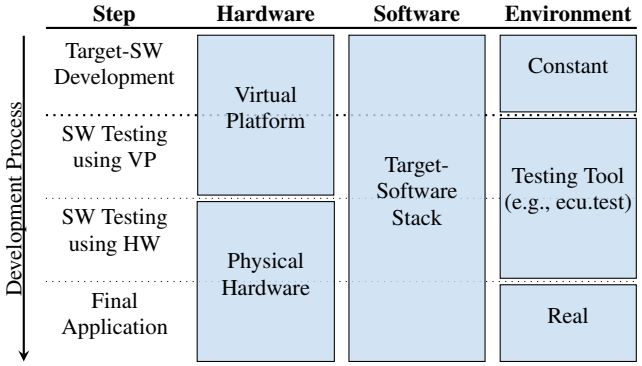


Figure 13. Incorporation of early target-software testing and validation in the design process.

Without our co-simulation integration, the test would have needed to be created directly for the physical hardware, making it possible only later in the development process.

When the target software stack passed all tests on the VP and the physical device, the application can be used in the real world application.

6 Conclusion and Future Work

In this paper, we have presented a novel approach to interface SystemC-based VPs with the FMI standard for co-simulation. By developing an adapter FMU that communicates with the VP via a TCP/IP-based connection, we enable the VP to exchange data with external tools in a co-simulation environment. Our method allows unmodified target software to run on the VP while interacting with realistic environmental inputs provided by other tools.

We demonstrated the effectiveness of our approach through a case study involving a Schmitt-trigger application running on the open-source AVP64. By connecting the VP to the FMI-compatible ecu.test tool, we supplied dynamic temperature inputs and observed the corresponding behavior of the software, achieving comprehensive testing and improved code coverage. This integration facilitates thorough software testing and verification by simulating environmental effects on the system, which is crucial for developing complex embedded systems.

While ecu.test is the leading testing tool in the automotive domain, and thereby a key candidate for evaluating our FMI integration, FMI opens the door to connect various other tools to VPs using our integration. Users have the possibility co-simulate VPs with models created by other tools, e.g., to add realistic physical environmental models to sensors of the VP. Tool providers that support FMI can directly benefit from our integration without the need of extending their tool. Moreover, VP developers can adopt a co-simulation approach to introduce physical environmental models when necessary, while retaining the flexibility to revert to a VP-only simulation if those models are not required.

Since our method only relies on VCML's VSP to establish the TCP/IP connection, it is not limited to a specific VP. There are no changes required for the VP to work with FMI, so other VPs can directly be integrated into an FMI co-simulation.

We showcased how the approach can be integrated into the target-software design process to enable early test development and testing. This speeds up the design process and reduces the number of errors in the product.

For future work, we plan to extend our approach to support more advanced communication between the VP and other FMUs by adding support for the new FMI Layered Standard for Network Communication (FMI-LS-BUS). This addition will allow connecting FMUs via virtual network interfaces like CAN or Ethernet.

References

Blochwitz, Torsten et al. (2011-06). "The Functional Mockup Interface for Tool independent Exchange of Simulation Models". en. In: pp. 105–114. DOI: 10.3384/ecp11063105. URL: https://ep.liu.se/en/conference-article.aspx?series=ecp&issue=63&Article_No=13.

Boehm, Barry W. (1976-12). "Software Engineering". In: *IEEE Transactions on Computers* C-25.12, pp. 1226–1241. ISSN: 1557-9956. DOI: 10.1109/TC.1976.1674590. URL: https://ieeexplore.ieee.org/document/1674590.

Bosbach, Nils et al. (2024-01). "NQC²: A Non-Intrusive QEMU Code Coverage Plugin". en. In: *Proceedings of the 16th Workshop on Rapid Simulation and Performance Evaluation for Design*. Munich Germany: ACM, pp. 16–21. ISBN: 9798400717918. DOI: 10.1145/3642921.3642924. URL: https://dl.acm.org/doi/10.1145/3642921.3642924.

Buildroot (2024). *Making Embedded Linux Easy*. URL: https://buildroot.org/.

Centomo, Stefano, Julien Deantoni, and Robert De Simone (2016-08). "Using SystemC Cyber Models in an FMI Co-Simulation Environment: Results and Proposed FMI Enhancements". In: *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 318–325. DOI: 10.1109/DSD.2016.86. URL: https://ieeexplore.ieee.org/document/7723569.

Free Software Foundation, Inc. (2024). *Remote Protocol (Debugging with GDB)*. URL: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Protocol.html.

IEEE Standards Association and others (2023-09). "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pp. 1–618. DOI: 10.1109/IEEESTD.2023.10246125.

ISO/TC 22/SC 32 (2018). *Road vehicles — Functional safety*. Geneva, Switzerland. URL: https://www.iso.org/standard/68383.html.

Ivanković, Marko et al. (2019-08). "Code coverage at Google". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, pp. 955–963. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3340459. URL: https://dl.acm.org/doi/10.1145/3338906.3340459.

Jünger, Lukas (2023-05). *An ARMv8 Virtual Platform (AVP64)*. URL: https://github.com/aut0/avp64.

Jünger, Lukas et al. (2019-01). "Fast SystemC Processor Models with Unicorn". In: *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*. RAPIDO '19. New York, NY, USA: Association for Computing Machinery,

pp. 1–6. ISBN: 978-1-4503-6260-3. DOI: 10.1145/3300189.3300191.

Kelly J. Hayhurst et al. (2001). *A Practical Tutorial on Modified Condition/Decision Coverage*. en. DIANE Publishing. ISBN: 978-1-4289-9599-4.

Krammer, Martin et al. (2015-09). "Standard compliant co-simulation models for verification of automotive embedded systems". In: *2015 Forum on Specification and Design Languages (FDL)*, pp. 1–8. URL: https://ieeexplore.ieee.org/document/7306083.

Linux Test Project (2023-10). *LTP GCOV extension (LCOV)*. URL: https://github.com/linux-test-project/lcov.

MachineWare (2024-03). *machineware-gmbh/vcml*. URL: https://github.com/machineware-gmbh/vcml.

Maxim Integrated (2015). *MAX31855 Datasheet*. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/max31855.pdf.

Miller, Joan C. and Clifford J. Maloney (1963-02). "Systematic mistake analysis of digital computer programs". In: *Commun. ACM* 6.2, pp. 58–63. ISSN: 0001-0782. DOI: 10.1145/366246.366248. URL: https://dl.acm.org/doi/10.1145/366246.366248.

Modelica Association (2010a-10). *Functional Mock-up Interface for Co-Simulation*. URL: https://fmi-standard.org/assets/releases/FMI_for_CoSimulation_v1.0.pdf.

Modelica Association (2010b-01). *Functional Mock-up Interface for Model Exchange*. URL: https://fmi-standard.org/assets/releases/FMI_for_ModelExchange_v1.0.pdf.

Modelica Association (2024-11). *FMI 3.0.2*. en. URL: https://github.com/modelica/fmi-standard/releases.

Modelica Association and MODELISAR Consortium (2024-11). *Functional Mock-up Interface Specification 3.0.2*. URL: https://fmi-standard.org/docs/3.0.2/.

Ondrej Burkacky et al. (2018-02). "Rethinking car software and electronics architecture". en. In: URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/rethinking-car-software-and-electronics-architecture#/.

Safar, Mona et al. (2018-05). "Virtual Electronic Control Unit as a Functional Mockup Unit for Heterogeneous Systems". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351357. URL: https://ieeexplore.ieee.org/document/8351357.

Saidi, Salah Eddine et al. (2019). "Fast Virtual Prototyping of Cyber-Physical Systems using SystemC and FMI: ADAS Use Case". In: *Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP'19)*. RSP '19. New York, NY, USA: Association for Computing Machinery, pp. 43–49. ISBN: 978-1-4503-6847-6. DOI: 10.1145/3339985.3358488. URL: https://dl.acm.org/doi/10.1145/3339985.3358488.

tracetronic GmbH (2024). *ecu.test*. URL: https://www.tracetronic.com/products/ecu-test/.