A Generic Protocol Approach for Integrating Heterogeneous Simulation Systems in Co-Simulation

Anonymous Authors
AnonymousEmail

***Abstract*—The validation of complex railway systems increasingly requires the integration of multiple specialized simulation tools that were developed independently using different technologies, programming languages, and interface conventions. This paper presents a generic protocol-based approach for bridging such heterogeneous simulation systems within a unified co-simulation framework. We introduce an abstraction to OpenMCx, that enables type-safe data exchange through dynamically discovered channels, eliminating the need for tool-specific adapters at the protocol level. The code is publicly available on GitHub. The resulting framework enables configuration-driven integration without requiring modifications to the participating systems, supporting both intellectual property protection and flexible system composition. We describe the component architecture, including the orchestrator's role, the generic gRPC-based communication component, the Python middleware implementation, and the adapter pattern for external simulator integration. Further, we presents extensibility patterns for incorporating new simulation tools and data types. The architecture has been validated through application to railway perception testing, demonstrating its effectiveness for practical validation scenarios.[1]**

***Index Terms*—co-simulation, testing, fmu, railway**

I. Introduction

Modern railway systems incorporate increasingly sophisticated automation, from advanced train control systems to autonomous operation capabilities [1]. Validating these systems requires simulation environments that can represent the complex interactions between vehicles, infrastructure, signaling, and perception systems. Simulation tools available for each of these domains were typically developed in isolation, using different programming languages, runtime environments, and communication paradigms. Integrating these tools into a unified environment presents significant technical challenges [2].

Current standardized co-simulation approaches such as Functional Mock-up Interfaces (FMI) [3] or the Robot Operating System (ROS) [4] require large effort to prepare a custom component for integration into a particular co-simulation setup. FMI, for example, requires a special packaging that freezes the interface definition at compilation time, and requires re-packaging if a new communication channel is needed. This especially occurs if a new agent is added to a simulation, and its state information shall be broadcasted to other (FMI-packaged) components. In ROS, adding new communication paths is easier, but the loose coupling of ROS nodes makes deterministic timing very challenging [5].

This paper and our extension to OpenMCx [6] presents a lightweight alternative approach based on a generic communication protocol that abstracts the details of data exchange into a uniform representation. Rather than developing adapters tailored to specific system pairs, we propose a generic message format that can represent arbitrary simulation variables regardless of their origin or type. Systems communicate through named channels that are discovered at runtime from configuration files, enabling new connections to be established without code changes. The approach leverages gRPC [7] for efficient cross-language communication and Protocol Buffers [8] for type-safe serialization, providing a solid technical foundation for heterogeneous system integration. We demonstrate the practical application of these concepts through a railway co-simulation case study.

The remainder of this paper is organized as follows. Section II reviews related work in co-simulation and give background information. Section III presents the technical approach, including the protocol design and channel discovery mechanism as well as the middleware architecture and its implementation. Section IV presents the railway case study and evaluation results, and Section V concludes with directions for future work.

II. Background information and related work

The challenge of integrating heterogeneous simulation systems has received considerable attention in the research community, leading to the development of several standards and frameworks that inform our approach.

A. Functional Mock-up Interface

The Functional Mock-up Interface (FMI) has emerged as the dominant standard for simulation model exchange and co-simulation in the industrial and scientific projects as well as in the automotive industry [3]. Originally developed through the MODELISAR project and now maintained by the Modelica Association, FMI defines a C-based application programming interface through which simulation models expose their capabilities to co-simulation masters. Models packaged according to this standard are called Functional Mock-up Units (FMUs) and can be executed by any compliant master algorithm.

[1]This version of the contribution has been accepted for publication, after peer review but is not the version of record and does not reflect post-acceptance improvements, or any corrections. The version of record will be linked here once available.

FMI version 2.0 supports two primary use cases: Model Exchange, where the master provides the numerical solver, and Co-Simulation, where each FMU includes its own solver and the master coordinates data exchange at discrete synchronization points. The Co-Simulation variant is particularly relevant to our work, as it enables the integration of models with vastly different internal implementations. However, FMI assumes that all participating models can be packaged as FMUs with C-callable interfaces, which may not be practical for legacy systems or tools implemented in managed runtime environments.

The recently finalized FMI 3.0 specification [9] extends the standard with features including support for discrete-time and triggered events, array variables, and terminal connections. These extensions address some limitations of the earlier specification but do not fundamentally change the requirement for C-based interfaces.

B. System Structure and Parameterization

Complementing FMI, the System Structure and Parameterization (SSP) standard [10] defines how co-simulation configurations are specified in a tool-independent manner. SSP uses XML-based formats to describe which components participate in a simulation, how their input and output variables are connected, and what parameter values should be applied. The System Structure Description (SSD) format captures the structural aspects of a co-simulation setup, while related formats address parameterization and result mapping.

SSP enables the same co-simulation configuration to be executed by different master tools without modification, promoting interoperability and reducing vendor lock-in. Our approach leverages SSP as the configuration mechanism for channel discovery, extending its use beyond traditional FMU-based co-simulation to encompass external system integration.

C. Robot Operating System

The Robot Operating System (ROS) [11] is a widely adopted middleware framework for multi-agent simulation that provides a loosely coupled publish-subscribe communication model, hardware abstraction, and a rich ecosystem of reusable components. While this architecture excels at modular robot software development, it is fundamentally at odds with the requirements of synchronized co-simulation. ROS communicates asynchronously among nodes, creating an event-based system in which each component operates at its own frequency with no global coordination of time steps [12]. The framework's own designers acknowledge that this asynchrony makes deterministic execution more difficult to achieve. By contrast, FMI-based co-simulation requires a central master algorithm to advance all participants in lockstep and to exchange data at well-defined synchronization points [3], a coordination primitive that ROS does not provide.

D. gRPC and Protocol Buffers

Google's gRPC framework [7] provides a modern foundation for cross-language remote procedure calls, using Protocol Buffers [8] for interface definition and message serialization. Unlike earlier RPC technologies, gRPC generates idiomatic client and server code for multiple programming languages from a single interface definition, significantly reducing the effort required to implement cross-language communication. Protocol Buffers offer efficient binary serialization with strong typing and forward compatibility, making them well-suited for simulation data exchange where both performance and type safety matter. Several recent co-simulation projects have adopted gRPC for external system integration, though typically with tool-specific message formats rather than the generic approach we propose.

E. Railway Simulation

Simulation-based validation in the railway domain traditionally focuses on specific aspects such as vehicle dynamics, infrastructure capacity, or signaling behavior. Efforts to combine these perspectives into integrated simulation environments have relied on custom integration code that limits reusability and maintainability. Recent work on scenario-based testing frameworks for highly automated railway systems in open context has demonstrated the value of combining open-source train simulators with formal test methodologies [1], [13].

Kugu et al. [14] present an FMI- and SSP-based model integration approach for railway digital twins, showing the applicability of established co-simulation standards to the railway domain. Our work builds on these foundations by providing a systematic integration approach that enables flexible composition of railway simulation capabilities. Unlike approaches that require all participants to be packaged as FMUs [3], [14], our gRPC-based protocol allows integration of systems that cannot easily expose C-callable interfaces, such as applications built on managed runtime environments.

F. OpenMCx Orchestrator

The orchestration function is provided by OpenMCx[6], an open-source co-simulation master that implements both FMI and SSP standards. OpenMCx reads the co-simulation configuration from SSP files, instantiates the specified participants, establishes data connections between them, and executes the co-simulation according to a master algorithm.

The orchestrator's configuration specifies the simulation structure through SSP's System Structure Description format. This includes the list of participating components, the connections between their input and output variables, and the parameter values that configure their behavior. OpenMCx parses this configuration at startup and uses it to guide the co-simulation execution.

III. Technical Approach

This section presents the core technical concepts underlying our integration approach: an application-agnostic gRPC service definition for generic data exchange, the channel discovery mechanism for configuration-driven connections, and the protocol definition for coordinated execution. The code is publicly available with the implemented changes that allow

co-simulation with a train simulator, e.g. OpenRails (Code repository omitted for double-blind review). Note, that the balise FMU which was used in our case-study in Section IV is not part of the release.

A. Application-agnostic gRPC Service Definition

Fig. 1 visualizes the specification of the gRPC service definition. The structure is intentionally simplistic. At simulation startup, the orchestrator sends a *InitializeRequest* (via the *Initialize* method) to the application-specific gRPC server. This signals start of the simulation run and communicates the expected step size and values for the component's parameters. In each simulation step, the *DoStep* method is called, sending component inputs and the actual step size to the server, and receiving component outputs in the response.

Central to our approach is the recognition that simulation data exchange, regardless of the specific domain or tool, ultimately reduces to the transfer of named values between components. The *VariableBinding* message is the fundamental unit of data exchange. Each binding associates a channel identifier with a typed value, enabling arbitrary simulation variables to be communicated through a uniform representation. The channel field contains a string identifier that uniquely names the variable within the co-simulation context. The value field uses Protocol Buffers' *oneof* construct to support multiple value types through a single message definition. Supported types include double-precision floating point for physical quantities, 32-bit integers for identifiers and counters, booleans for flags and states, strings for textual data, and binary bytes for complex structures. This type diversity accommodates the range of data commonly exchanged in simulation scenarios. The protocol layer handles *VariableBinding* messages without understanding what the values mean. That interpretation occurs at the application layer where domain knowledge resides. This separation enables the protocol infrastructure to be genuinely generic.

B. Dynamic Channel Discovery

A key principle of our approach is that the connections between simulation components should be defined in configuration rather than code. This enables the co-simulation topology to be modified without recompiling any participant, supporting rapid experimentation and flexible system composition.

Channel discovery occurs at initialization time by parsing SSP [10] configuration files. The System Structure Description specifies which components participate in the co-simulation and how their connectors are linked. OpenMCx parses this specification and builds an internal registry mapping channel names to their sources and destinations. When a component sets a value on a channel, the registry routes that value to all connected inputs; when a component requests a channel's value, the registry provides the most recent value from the connected output. This configuration-driven approach extends naturally to external systems integrated through gRPC. Adding a new external system to an existing co-simulation requires only updating the configuration file and ensuring the new system implements the expected protocol.

C. System Architecture Overview

The gRPC component compiled into OpenMCx is fully generic and reusable. It can work with any backend, because OpenMCx handles component connections independently. Any gRPC server, as long as it implements our `GRPCComponentService` protocol, and any variable names/types, because they're defined dynamically in the SSD file. More details on that in subsection III-E.

The middleware layer exists to bridge the gap between the internal participant interface and external systems with different communication paradigms. This layer handles protocol translation, time model adaptation, and data format conversion. The middleware isolates the orchestrator from the complexity of external integration, presenting a clean interface while managing the details of external communication internally.

The external simulator layer comprises the actual simulation tools that provide domain-specific capabilities. These tools may be interactive applications, command-line programs, network services, or any other form of simulation capability. The middleware layer adapts each external simulator to participate in the co-simulation without requiring modifications to the simulator itself.

Data flows through the architecture in patterns determined by the co-simulation configuration and orchestrator algorithm. In the most common pattern, the orchestrator initiates each simulation step by providing input values to all participants. Each participant then advances its internal simulation by the specified time increment, computing new output values based on the inputs and its internal state. The orchestrator collects these outputs and routes them to the appropriate inputs for the next step.

For participants implemented as native FMUs, this data flow occurs through direct function calls following the FMI specification. For external systems, the data flow traverses additional layers. The orchestrator calls the component implementation through the generic gRPC interface. The component may either perform computations directly, or act as a middleware that translates this to application-specific messages sent to external systems (Fig. 2 shows the latter case). The external systems process the request and return responses. The middleware sends these back to orchestrator.

This multi-layer data flow introduces latency compared to native FMU communication, but the latency remains acceptable for simulation scenarios where the computation time of the actual simulation dominates the communication overhead. Also, the overhead introduced through gRPC is minimal (if running on the same machine) because inter-process communication (e.g. named-pipes, shared memory) can be used. The architectural separation provides flexibility and maintainability benefits that outweigh the modest performance cost.

D. Application specific gRPC Server

In our case-study (see Section IV) the application specific gRPC Server connects OpenMCx to the OpenRails train sim-

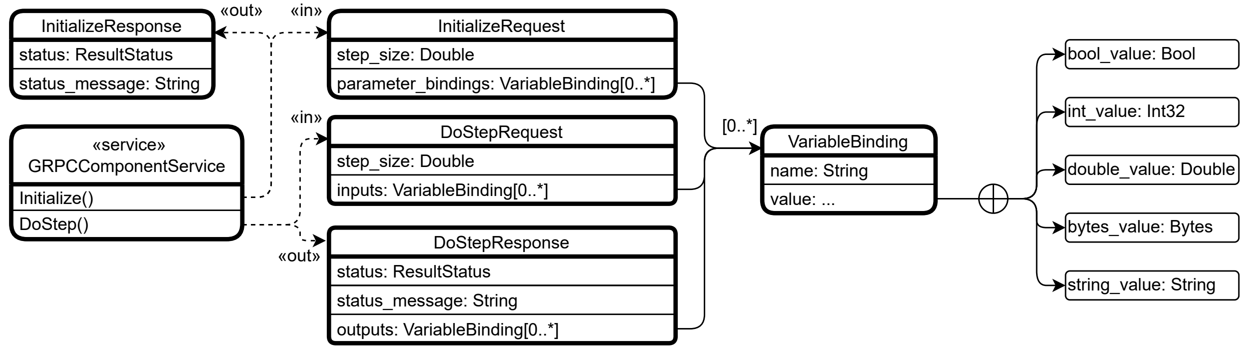


Fig. 1: `GRPCComponentService` definition

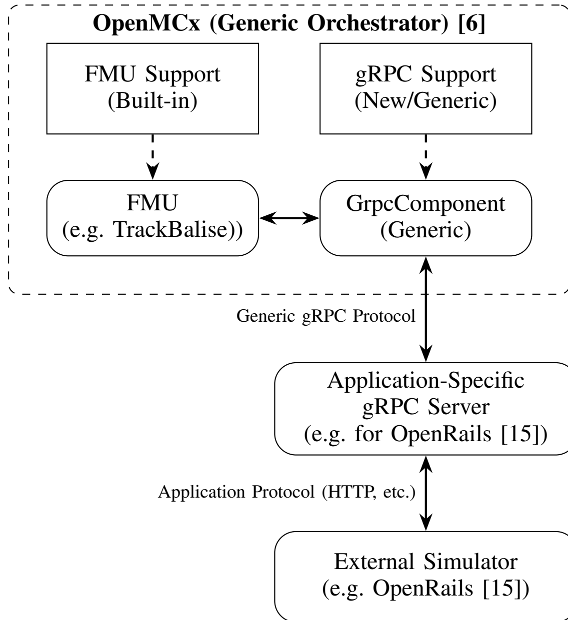


Fig. 2: Architecture of our approach. The rectangles depict general support of the respective technology, the boxes with rounded corners conrete instances.

ulator. This adapter exemplifies the pattern used for external simulator integration as detailed in subsection III-E. OpenRails provides simulation state and accepts control inputs through an HTTP API [15]. The server maintains an HTTP session with the running OpenRails instance, periodically polling for state updates and applying control commands as requested by the middleware. State retrieval extracts relevant values from OpenRails' responses, including train position, speed, and control system status. These values are converted to the appropriate units and formats expected by other co-simulation participants. For example, OpenRails may report position in terms of track section and offset, while other participants expect a continuous distance value; the adapter performs this transformation. Control application translates abstract control requests into specific OpenRails commands. A throttle setting from the co-simulation becomes an HTTP request to the appropriate OpenRails control endpoint. The adapter handles the details of command formatting and response verification, presenting a clean interface to the middleware layers above. The server also manages the time relationship between co-simulation steps and OpenRails' continuous simulation. OpenRails normally runs in real time or accelerated real time. The adapter controls this rate to synchronize with co-simulation requirements, pausing and resuming the simulator as needed to maintain consistent time across all participants.

E. Extensibility Patterns

The architecture supports extension in several dimensions: adding new external simulators and introducing new data types. This section presents patterns for each extension type.

1) New External Simulators: Integrating a new external simulator follows a consistent pattern. First, the simulator's capabilities and interface are analyzed to understand what state it can provide and what controls it accepts. Second, an adapter class is implemented that translates between the middleware's generic interface and the simulator's specific API. Third, channel mappings are defined that associate simulator variables with co-simulation channels. Fourth, configuration elements are added to include the new simulator in co-simulation scenarios.

The adapter class implements a standard interface expected by the Application Layer. This interface includes methods to establish connection with the simulator, retrieve current state as a dictionary of values, apply input values from a dictionary, and clean up on shutdown. The Application Layer calls these methods at appropriate points during co-simulation, without needing to understand the specific simulator being adapted. This pattern isolates simulator-specific complexity within the adapter, keeping the middleware core generic. Multiple adapters can coexist, enabling co-simulation scenarios that combine capabilities from diverse simulation tools.

2) New Data Types: The *VariableBinding* message's support for binary data enables extension to complex data types beyond the primitive types directly supported. When a simulation requires structured data, e.g. geometric coordinates, signal aspects, train consist descriptions, these can be serialized to binary format and transmitted through the standard protocol.

The recommended approach defines additional Protocol Buffer messages for complex types, using standard serialization to convert them to the binary format expected by *VariableBinding*. Receiving components deserialize back to the structured representation. This pattern provides type safety for complex data while maintaining compatibility with the generic protocol. For frequently used complex types, future protocol versions might incorporate them as value alternatives, eliminating the serialization overhead. The current design prioritizes flexibility and backward compatibility over optimization for specific data types.

IV. Case Study: Railway Perception Testing

To validate the approach, we applied it to a railway perception testing scenario [16] involving a balise-based train localization system. Balises are passive transponders installed in the track that transmit their identity and location as well as information about upcoming signaling when a train antenna passes overhead. They are part of the European Train Control System (ETCS) that is currently rolled out to standardize train protection systems across Europe [17]. The train uses balise detection, combined with odometry, to determine its position along the route. Validating this system requires simulation of train dynamics, balise infrastructure, and the detection process. Therefor the integration of previously incompatible systems is about to be demonstrated.

A. System Implementation

A custom TrackBalises component (FMU) models the balise infrastructure and detection logic. It is included in Fig. 2 as an example. Infrastructure data, including the balise locations, has been provided by DB InfraGO for the actual track section Annaberg-Buchholz to Schwarzenberg as a XML-file. This XML-file is processed in the TrackBalises component by extracting the geographic coordinate points, calculating a continuous track consisting of B-Splines and associating the balise ID to traveled distance on the selected track section. During simulation it receives the current position along track as an input and returns the balise ID that is supposed to be received last. Visualization in OpenRails verifies, that indeed the balise ID in figure Fig. 3 is updated whenever the train passes a yellow block that represents a balise as shown in Fig. 4. The model is based on former work of Heckmann et al. [18], extending the Track module of Modelica RailwayDynamics Library [19]. It was created as a Modelica Model in Dymola and later exported as FMU.

Prior to this work, the TrackBalises component could not interoperate with OpenRails via OpenMCx (or to our knowledge any other co-simulation orchestrator). OpenRails was designed as a standalone interactive application with no external control interface. OpenMCx expected all participants to be FMUs with C-callable interfaces.

B. Integration Implementation

Applying our approach, we developed three integration elements. First, we extended OpenRails with a Python frontend, transforming it from an interactive application into a controllable simulation component [15]. The frontent communicates with the OpenRails simulator core via an HTTP API enabling external control and state access. Second, we extended the frontend to a three-layer middleware, handling gRPC communication with OpenMCx and HTTP communication with OpenRails. Third, we extended OpenMCx (see Section III) so it handles gRPC-connected components like any other simulation component.

The SSP configuration file defines the co-simulation structure, specifying the connections between OpenRails position output, TrackBalises position input, and the resulting balise detection outputs. This configuration can be modified to add additional participants or change connection topology without rebuilding any component.

C. Results

The integrated system successfully executes co-simulation scenarios where the train traverses a route containing multiple balises. Position data flows from OpenRails through the middleware to the balise detection logic, which correctly identifies balise crossings and reports them back through the co-simulation outputs. A script automatically creates the OpenRails format of the map, using the XML-file, described above, as input. For a better visualization of the co-simulation we created a map-server web application using Flask [20] and Dash [21] (see Fig. 3) which runs in a browser. Flask serves as a bridge, processing the incoming gRPC server requests - containing updates of cosimulation results - to dynamically refresh both the map visualization and the balise readings tabular data. The map interface functions as a real-time progression monitor, visualizing the train's precise coordinates synchronized with the OpenRails train position (see Fig. 4).

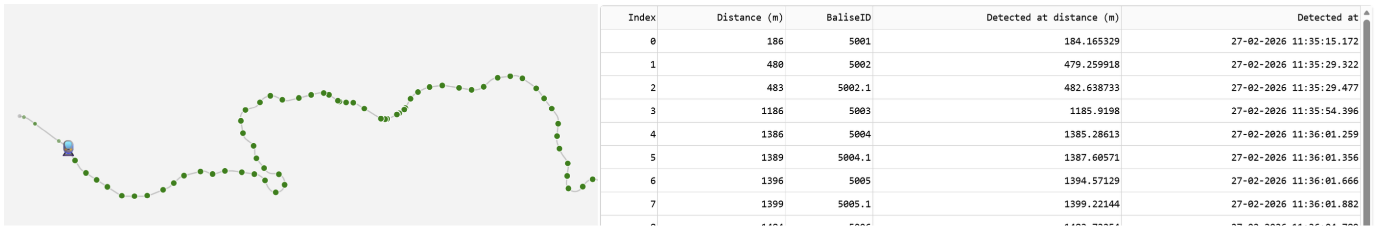
The configuration-driven nature of the integration proved valuable during development. As the co-simulation topology evolved, connections could be added and modified through configuration file changes without rebuilding components. This flexibility significantly accelerated the integration process compared to traditional hard-coded approaches.

V. Conclusion

This paper has presented the architecture of a middleware system for railway co-simulation, describing the component structure, communication protocol, configuration approach, and extensibility patterns.

The key architectural contributions include the generic gRPC component that allows a seamless integration of external tools, enabling their participation in coordinated co-simulation scenarios without requiring modifications to the tools themselves, and the configuration-driven approach that enables flexible system composition. The *VariableBinding* abstraction enables type-safe data exchange through dynamically discovered channels, while our architecture cleanly separates orchestration, protocol, and application concerns.

The approach contributes to the broader goal of simulation-based validation for complex systems by reducing the integra-



Index	Distance (m)	BaliseID	Detected at distance (m)	Detected at
0	186	5001	184.165329	27-02-2026 11:35:15.172
1	480	5002	479.259918	27-02-2026 11:35:29.322
2	483	5002.1	482.638733	27-02-2026 11:35:29.477
3	1186	5003	1185.9198	27-02-2026 11:35:54.396
4	1386	5004	1385.28613	27-02-2026 11:36:01.259
5	1389	5004.1	1387.60571	27-02-2026 11:36:01.356
6	1396	5005	1394.57129	27-02-2026 11:36:01.666
7	1399	5005.1	1399.22144	27-02-2026 11:36:01.882

Fig. 3: A screenshot of a rendering of the route, the train on it, and the balises (green dots). The table on the right updates with the detected traveled distance, ID, and time when a balise is passed.

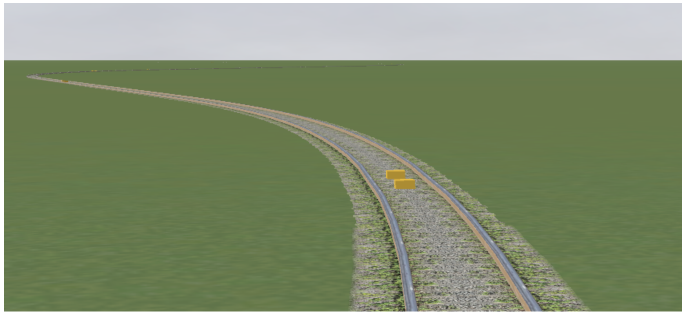


Fig. 4: Output of RGB front-camera sensor from OpenRails. Two balises are shown in yellow on the track.

tion burden that often impedes effective use of available simulation capabilities. Rather than developing custom adapters for each tool combination, organizations can implement the generic protocol once and configure connections as needed for specific validation scenarios. An application to a railway case-study demonstrated the practical value of the approach, successfully integrating previously incompatible systems without requiring modifications to their implementations.

Future architectural evolution will address additional requirements as they emerge. Support for distributed deployment, where middleware components run on separate machines from the orchestrator, will enable scaling to larger scenarios. Integration with additional orchestration frameworks beyond OpenMCx will demonstrate architectural generality. Enhanced monitoring and debugging capabilities will support operation of complex co-simulation configurations. The architecture presented here provides a sound foundation for these extensions while meeting current requirements for railway co-simulation validation.

REFERENCES

[1] M. Wild, J. S. Becker, G. Ehmen, and E. Möhlmann, "Towards Scenario-Based Certification of Highly Automated Railway Systems," in *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, 2023.

[2] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: A survey," *ACM Computing Surveys*, 2018.

[3] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proceedings of the 9th International Modelica Conference*, 2012.

[4] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science robotics*, vol. 7, no. 66, 2022.

[5] S. Sagmeister, M. Weinmann, P. Pitschi, and M. Lienkamp, "RSLCPP: Deterministic simulations using ROS 2."

[6] Eclipse Foundation, "OpenMCx: Open co-simulation middleware," accessed: 2026. [Online]. Available: https://projects.eclipse.org/projects/technology.openmcx

[7] Google, "gRPC: A high performance, open source universal RPC framework," 2015, accessed: 2026. [Online]. Available: https://grpc.io/

[8] ——, "Protocol buffers: Language-neutral, platform-neutral extensible mechanism for serializing structured data," 2008, accessed: 2026. [Online]. Available: https://protobuf.dev/

[9] Modelica Association, "Functional mock-up interface for model exchange and co-simulation," 2026. [Online]. Available: https://fmi-standard.org/

[10] ——, "System structure and parameterization, version 1.0," 2026. [Online]. Available: https://ssp-standard.org/

[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," *ICRA Workshop on Open Source Software*, 2009.

[12] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, 2022.

[13] M. Wild, J. Becker, C. Schneiders, and E. Möhlmann, "A Scenario-Based Simulation Framework for Testing of Highly Automated Railway Systems," in *Proceedings of the 11th International Conference on Vehicle Technology and Intelligent Transport Systems*, 2025.

[14] O. Kugu, S. Zhou, R. Nowak, G. Müller, S. H. Reiterer, A. Meierhofer, S. Lachinger, L. Wurth, and M. Grafinger, "An FMI- and SSP-based Model Integration Methodology for a Digital Twin Platform of a Holistic Railway Infrastructure System," in *Proceedings of the 15th International Modelica Conference*, 2023.

[15] (Author omitted for double-blind review), "(title omitted for double-blind review)," accessed: 2026. [Online]. Available: (Coderepositoryomittedfordouble-blindreview)

[16] M. Wild, J. S. Becker, A. Buinoschi-Tirpescu, and E. Möhlmann, "Towards Virtual Testing of Perception in the Railway Domain," in *IEEE International Automated Vehicle Validation Conference (IAVVC)*, 2025.

[17] European Union Agency for Railways, "ERTMS/ETCS system requirements specification," ERA, Tech. Rep., 2016, subset-026.

[18] A. Heckmann, A. Poßeckert, and V.-B. Adusumalli, "Aspects and ideas for the fmi-based modeling of railway digital twins," in *Proceedings of 16th International Modelica & FMI Conference*. Modelica Association and Linköping Electronic Conference Proceedings, 2025.

[19] A. Heckmann, M. Ehret, G. Grether, A. Keck, D. Lüdicke, and C. Schwarz, "Overview of the DLR RailwayDynamics Library," in *Proceedings of the13th International Modelica Conference*, 2019.

[20] Pallets, "Flask," 2010. [Online]. Available: https://flask.palletsprojects.com/

[21] C. Parmer, P. Duval, and A. Johnson, "A data and analytics web app framework for Python, no JavaScript required," 2024. [Online]. Available: https://github.com/plotly/dash/