Journal Title
XX(X):1–24

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Adapting Parallel DEVS Semantics to FMI 3.0 Co-Simulation Using Synchronous Clocks

Yon Vanommeslaeghe[1,2], Claudio Gomes[3], Bert Van Acker[1,2], Joachim Denil[1,2], Paul De Meulenaere[1,2]

Abstract

Simulating cyber-physical systems (CPS) often requires integrating continuous dynamics, typically described by differential equations, with discrete-event dynamics that represent the behavior of embedded controllers. Version 3.0 of the Functional Mock-up Interface (FMI) standard introduces synchronous clocks and a discrete-event mode, creating new opportunities for the co-simulation of hybrid systems. So far, it has not been demonstrated that these features can faithfully capture the semantics of discrete-event formalisms such as the Discrete EVent System specification (DEVS) and its extension, Parallel DEVS (PDEVS). This leaves open the question of whether FMI 3.0 can effectively support complex hybrid dynamics.

This paper addresses this gap by showing how PDEVS models can be exported and coupled as FMI 3.0 Functional Mock-up Units (FMUs) while preserving PDEVS semantics. We first derive requirements that any simulator must satisfy to correctly preserve PDEVS semantics. Based on these requirements, we present a systematic mapping of PDEVS abstract simulator functions to the FMI 3.0 interface with synchronous clocks and present an orchestration algorithm to co-simulate multiple PDEVS FMUs accordingly. The approach is validated using an adaptive cruise control case study, where our FMI-based co-simulation reproduces the results of an established PDEVS simulator. To support reproducibility and further research, we provide an open-source prototype together with the complete validation setup.

Keywords

discrete event system specification, functional mock-up interface, co-simulation, cyber-physical systems

Introduction

In general, the simulation of Cyber-Physical Systems (CPS) requires simulating both continuous dynamics, which evolve smoothly over time, and discrete-event dynamics, where changes occur instantaneously at specific points in time. The *physical part* of a CPS is typically described using Ordinary Differential Equations (ODEs) to capture the underlying continuous dynamics. The *cyber part*, in contrast, is inherently discrete, with controllers, communication protocols, and embedded systems, which operate on clocks and both react to and produce events. The challenge in faithfully simulating CPS lies in coordinating these two fundamentally different domains in a coherent co-simulation framework. Over the years, various formalisms have been proposed. Broadly, these approaches can be classified as *discrete-event first*, where all subsystems are converted to a discrete-event formalism, and *continuous-time first*, where all subsystems are expressed in continuous time and discretization merely happens to enable stepwise simulation.

The Discrete EVent System specification (DEVS)[1] provides a well-defined methodology to model and simulate systems where state changes are driven by events occurring at discrete points in time. It is particularly effective in scenarios where the interactions between subsystems are naturally described as sequences of events, such as embedded systems and communication protocols. DEVS and its extension, Parallel DEVS[2] (PDEVS), are widely used for modeling large-scale discrete-event systems. These formalisms not only provide expressive modeling constructs, but also specify precise operational semantics. Moreover, many other modeling formalisms can be transformed into DEVS,[3,4] positioning it as a unifying framework to integrate different formalisms. This makes DEVS a natural candidate for modeling the cyber components of CPS and for capturing their interactions with the continuous physical dynamics.

The Functional Mock-up Interface (FMI) standard has become the de facto mechanism for enabling co-simulation across tools. Earlier versions of FMI, however, were designed around a continuous-time first perspective, lacking explicit mechanisms for accurate event detection and localization, which limited interoperability with discrete-event formalisms such as PDEVS. Version 3.0 of FMI introduces new features that open opportunities to address this gap. More specifically, it introduces the concept of *synchronous clocks*, which allow subsystems to synchronize on common

[1]Cosys-Lab (FTI), University of Antwerp, Belgium
[2]AnSyMo/Cosys, Flanders Make, Belgium
[3]Aarhus University, Denmark

Corresponding author:
Yon Vanommeslaeghe
Email: yon.vanommeslaeghe@uantwerpen.be

times, and the *discrete-event mode*, facilitating the propagation of discrete events interleaved with continuous dynamics. These extensions open the door to faithfully capturing discrete-event behavior within FMI-based co-simulation.

However, while the semantics of synchronous clocks have been defined,[5,6] and an orchestration algorithm has been proposed by Hansen et al.,[7] there *has been no work to date demonstrating the ability of FMI 3.0 FMUs to faithfully capture the semantics of discrete-event formalisms, such as PDEVS*. This leaves open the question of whether FMI 3.0 can effectively simulate complex hybrid dynamics where discrete-event and continuous subsystems interact.

In this paper, we take a first step toward this goal by focusing on the discrete-event side. Our central question is whether FMI 3.0 can be used to faithfully co-simulate PDEVS models when exported as FMUs. This is nontrivial, as PDEVS not only defines a modeling formalism but also precise operational semantics for how coupled models should interact: how internal and external transitions are scheduled, how simultaneous events are handled, and how outputs are routed between components. When PDEVS models are exported as FMUs, these semantics must be preserved, otherwise, the coupled FMUs would no longer reproduce the behavior defined by the PDEVS formalism.

Contribution & Prior Work. Our previous work[8] did not extend to coupling multiple DEVS FMUs through the FMI interface. It demonstrated how FMI 3.0 can be used to co-simulate *a single DEVS model*, encapsulated as an FMU to evaluate the performance of embedded applications, but did not couple multiple DEVS models through FMI. That work leveraged the discrete-event mode and synchronous clocks to signal *the occurrence of events*, but events could not carry other data, as required in large scale DEVS simulations.

The present work addresses these limitations by demonstrating how PDEVS models can be faithfully exported and coupled as FMI 3.0 FMUs without violating the semantics of PDEVS. Note that in this paper, we adopt Parallel DEVS instead of "classic" DEVS. PDEVS resolves key limitations of DEVS, most importantly the handling of simultaneous events, which is essential in the context of FMI-based co-simulation. The rationale for this choice is discussed in detail in Section Challenges in Coupling Classic DEVS Models.

The contributions of this paper are fourfold:

1. We derive a *set of requirements* that must be satisfied to preserve PDEVS semantics when models are encapsulated as FMUs and coupled through FMI.
2. We present an *implementation outline* that defines the *systematic mapping* between PDEVS abstract simulator functions and the FMI 3.0 interface with synchronous clocks in a manner that satisfies the requirements.
3. We present an *example orchestration algorithm* to co-simulate multiple PDEVS FMUs using synchronous clocks and discrete-event mode to enable correct event scheduling, communication, and time advancement, further satisfying the requirements.
4. We *validate the approach* using an example case study, comparing our approach to an established PDEVS simulator. Simulation results confirm that PDEVS semantics are preserved in practice.

To foster reproducibility and further research, we also release an *open-source prototype implementation* together with the complete validation setup.

Structure. The remainder of this paper is organized as follows. Section Background first describes the adaptive cruise control case study that we use as a running example, and then introduces the necessary background on FMI 3.0 with synchronous clocks as well as the DEVS and PDEVS formalisms. Section Related Work positions our work in relation to prior research on integrating DEVS with other formalisms, such as FMI. Section Contribution presents our main contribution: the requirements for preserving PDEVS semantics, the mapping to FMI 3.0 functions, and an orchestration algorithm for PDEVS FMUs. In Section Results and Discussion, we evaluate the correctness of our approach against a PythonPDEVS reference simulation, interpret the results and highlight broader implications. Section Limitations discusses the main limitations of our approach and outlines potential ways to address them. Finally, Section Conclusions and Future Work summarizes the findings and outlines directions for future work.

Background

Running Example

To demonstrate the use of FMI synchronous clocks and PDEVS, we consider an example of an Adaptive Cruise Control (ACC) system, adapted from an example provided by the MathWorks.[9] An ACC system is designed to enhance vehicle safety and comfort by automatically adjusting the vehicle's speed. Unlike conventional cruise control, which maintains a constant speed set by the driver, an ACC dynamically adapts the *ego vehicle*'s speed based on traffic conditions by tracking a car in the front (i.e., the *lead vehicle*). The overall system architecture is shown in Figure 1. To simplify the illustration of our contribution, we consider three variants of the system, shown in the figure: the open loop scenario is used to illustrate how the event communication is implemented; the closed loop scenario is used to motivate the use of Parallel DEVS and how it is mapped to FMI, and helps illustrate our design choices for the orchestration of events; finally, the full scenario validates the semantic equivalence between PDEVS and the FMI-based co-simulation.

The ACC system operates in two primary modes: speed control and distance control. In speed control mode, it maintains a fixed target speed, typically active at higher speeds on highways. In distance control mode, it maintains a safe distance from the *lead vehicle*, and is typically active in congested traffic. A *supervisor* manages the mode transitions based on the current speed and relative distance to the *lead vehicle*. The supervisor updates the *controller*'s speed setpoint, setting it either to the maximum allowed speed or a reduced speed to maintain a safe following distance. The *controller* then adjusts the acceleration of the *ego vehicle* accordingly. Further implementation details are provided in Section PDEVS Models of the Running Example.

Test Scenario. To make the running example more concrete, we define a test scenario. In this scenario, the ego and lead vehicles begin with predefined initial conditions and

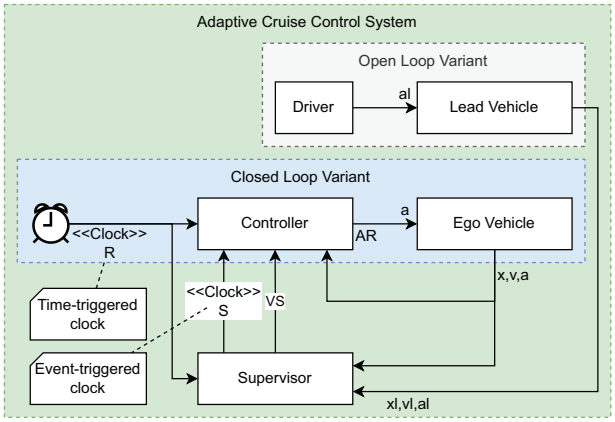

Figure 1. Running example of an adaptive cruise controller using synchronous clocks. The different clock types are defined in Section FMI 3.0 - Synchronous Clocks (SC).

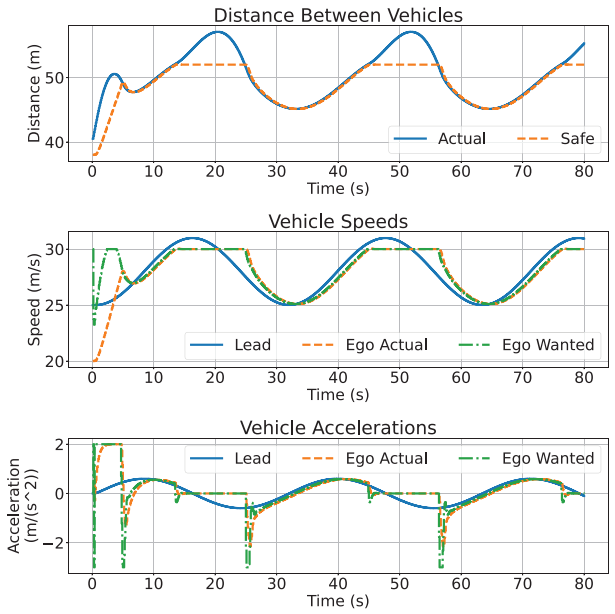

Figure 2. Traces obtained using PythonPDEVS[10] for the described test scenario.

interact with each other through the ACC system. In the test scenario, the *ego vehicle* starts out at *position* $10\,\mathrm{m}$, *velocity* $20\,\mathrm{m\,s^{-1}}$, *and acceleration* $0\,\mathrm{m\,s^{-2}}$, while the *lead vehicle* starts at *position* $50\,\mathrm{m}$, *velocity* $25\,\mathrm{m\,s^{-1}}$, *and acceleration* $0\,\mathrm{m\,s^{-2}}$. The driver causes the lead vehicle to accelerate and decelerate following a *sine wave* with *amplitude* $0.6\,\mathrm{m\,s^{-2}}$ *and* ω $0.2\,\mathrm{rad\,s^{-1}}$. The ACC system is tasked with maintaining a safe distance of $1.4\,\mathrm{s} * \mathit{ego\_vehicle\_speed} + 10\,\mathrm{m}$ between the two vehicles, with a maximum allowed speed for the ego vehicle of $30\,\mathrm{m\,s^{-1}}$.

Figure 2 presents simulation traces for this scenario. As can be seen, the system maintains the ego vehicle's speed at $30\,\mathrm{m\,s^{-1}}$ whenever the actual distance between the ego and lead vehicles exceeds the safe distance, such as during the interval from $13\,\mathrm{s}$ to $25\,\mathrm{s}$. However, as the ego vehicle closes in on the lead vehicle, its speed is reduced to maintain a safe distance, as seen between $25\,\mathrm{s}$ and $45\,\mathrm{s}$.

FMI 3.0 - Synchronous Clocks (SC)

Co-simulation involves integrating multiple simulation tools or models to analyze complex systems collaboratively.[4] To facilitate interoperability between different simulation environments, the Functional Mock-up Interface (FMI) standard defines a standardized wrapper for models (with their solvers), referred to as the Functional Mock-up Unit (FMU).[11] These FMUs expose a standardized interface, allowing external tools, known as importers, to integrate and interact with them in a tool-agnostic manner.

Earlier FMI versions primarily supported co-simulation of continuous dynamic systems. However, FMI 3.0 introduces significant enhancements, particularly in the domain of discrete event simulation. A key addition is the concept of synchronous clocks,[12] which enable precise synchronization of events across different FMUs. Here we introduce clocks informally and refer the reader to[5,13] for more formal treatments. These features are particularly relevant for integrating DEVS models within FMI-based co-simulations.

Motivation for Synchronous Clocks in Co-Simulation. To illustrate the need for SCs, consider the Adaptive Cruise Control (ACC) system in Figure 1. The ACC system can be modeled as a co-simulation of multiple FMUs, each representing a system component such as the controller, supervisor, driver, ego vehicle, or lead vehicle.

This system includes both continuous and discrete dynamics operating at different rates. The ego and lead vehicles exhibit continuous behavior, whereas the supervisor, controller, and driver operate in a discrete manner. The controller and supervisor execute periodic routines (e.g., every 10ms), though the supervisor may or may not change the controller's setpoint at each step. Meanwhile, the driver FMU can change its output at arbitrary times. Clocks are used to synchronize the execution of these routines within the different FMUs, e.g., ensuring that the controller and supervisor are always in sync.

To co-simulate the above scenario using FMI *without synchronous clocks*, where each FMU provides the *get*, *set*, and *doStep*, operations, the orchestration algorithm would follow the simulation loop:[14]

1. Advance simulated time.
2. Determine the next simulated synchronization time. In the above scenario, it could be current simulated time + 10ms, or the time when the driver FMU is expected to change its output.
3. Synchronize all FMUs to this simulated time by calling their *doStep* function with the same simulated time step (e.g., 10ms).
4. Exchange data between FMUs by calling their *get* and *set* functions.
5. Repeat steps 1-5 until the target simulated time has been reached.

This orchestration poses challenges in synchronizing routines:

- **Predicting driver output changes:** The orchestrator must be able to predict when the driver will change its output, which is not always straightforward. Broman et al.[15] identified this as a key challenge in co-simulation.

- **Timing drift due to floating-point arithmetic:** The periodic execution of the controller and supervisor FMUs may drift over time due to numerical inaccuracies in floating-point representation. This issue can only be avoided by using integer-based timing, fixed-point arithmetic, or designating a single FMU as the timekeeper for the simulation.

Synchronous Clocks and Event Mode in FMI 3.0. FMI 3.0 introduces SCs to address these synchronization issues. Another key addition in this version is the introduction of event mode, alongside the traditional step mode. Event mode allows FMUs to explicitly handle discrete events, such as input events, time events, and state changes, without advancing simulation time. This ensures that all discrete changes are resolved before proceeding with time stepping, improving synchronization accuracy. Additionally, FMUs can now signal early return from a *doStep* call, indicating that an event has occurred and must be handled before continuing. These enhancements provide greater flexibility in co-simulation by allowing fine-grained event handling in systems with mixed continuous and discrete behaviors.

Co-simulations with SC follow a structured execution sequence, incorporating step execution in continuous time mode (similar to traditional co-simulation), event detection, and event handling phases. Before we describe the orchestration algorithm, we first introduce the different types of clocks defined in FMI 3.0:

Time-based clocks trigger events at specific intervals during a simulation and can be categorized based on their behavior. For instance, periodic clocks trigger at constant intervals, while aperiodic clocks activate based on specific conditions rather than a fixed schedule. For other time-based clock types, we refer the reader to Gomes et al.[5]

Triggered clocks, in contrast with time-based clocks, are activated by state changes or external conditions, without a predefined interval. These include: *Input Triggered Clocks*, activated by the orchestrator to signal state or input changes; and *Output Triggered Clocks*, activated internally by an FMU based on its internal logic, signaling events to other FMUs or the orchestrator.

In the ACC example (Figure 1), the controller and supervisor both have input time-based clocks R that are set by the importer, while other clocks are event-triggered. For instance, the controller FMU has an input triggered clock S that is connected to the supervisor's output event clock with the same name.

Orchestration Algorithm for FMI 3.0 Synchronous Clocks. Ravi et al.[16] address the challenge of orchestrating simulations for FMI SC and provided the basis for the orchestration algorithm presented in this paper. The proposed orchestration algorithm divides the simulation process into a repetition of co-simulation steps that incrementally advanced the simulated time until it meets the stopping criteria. Each co-simulation step has three phases: step execution (step mode), event detection, and event handling (event mode), detailed next:

1. **Step execution (step mode):** the same as in traditional co-simulation. Step and synchronize FMU states by invoking the *doStep* function for all FMUs and exchanging data, and obtain the new simulation time. FMUs can return earlier from the *doStep* function if they have detected an event. FMUs are set to step mode.
2. **Event detection:** Compute the set of ticking clocks from time-based clocks, and detect events triggered by state-based conditions or active event clocks (triggered clocks).
3. **Event handling (event mode):** For all FMUs with active clocks:
   (a) Transition the FMUs to Event mode.
   (b) Resolve events by:
      i. Iterating on discrete equations and clocked variables while respecting their dependencies.
      ii. Propagating clock activation values between connected clocks across FMUs.
      iii. Exchange relevant data between FMUs in event mode.
   (c) Execute the discrete event step function for all FMUs in event mode.
   (d) Repeat event handling if necessary.
   (e) Return FMUs to *Step mode* and repeat.

Thanks to the use of synchronous clocks, the event detection and handling is carried out in a controlled manner and the FMU co-routines (which are run when their outputs are queried) are run synchronously, and numerical inaccuracies are circumvented by having a single source of clock ticking authority at all times.

In the following subsection, we first introduce the DEVS formalism as a foundation for modeling discrete-event systems, and then present its extension, Parallel DEVS.

Discrete Event System Specification (DEVS)

This subsection provides a brief overview of the relevant aspects of the discrete event system specification using a didactic example of a traffic light system. A more extensive discussion regarding the DEVS concepts can be found in the work by Vangheluwe.[17]

The Discrete EVent System specification (DEVS), introduced by Zeigler,[1] is a framework used to model and simulate systems where events occur at specific moments in time. It organizes system behavior into two levels: *atomic models* and *coupled models*.

Atomic DEVS Models. Atomic DEVS models are the basic building blocks that describe how a system behaves over time. They describe the behavior of a discrete-event system as a sequence of deterministic transitions between sequential states, including how it reacts to external input (events) and how it generates output (events).[17] Figure 3 shows an illustration of two such atomic DEVS models, one of a police officer ("Police"), and one of a traffic light ("TrafficLight").

Formally, an atomic DEVS model can be defined as:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle$$

With:

- S the *set of admissible sequential states*
- $ta : S \to \mathbb{R}^{+}_{0,+\infty}$ the *time advance function*, which models the time the system remains in a certain state, before transitioning to the next sequential state
  Notes: (1) instantaneous transitions can be modeled

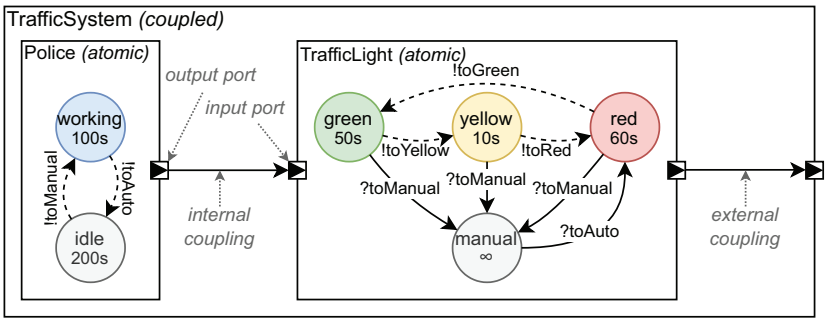

Figure 3. Illustration of a traffic system model, modeled using DEVS.[8]

using $ta(s) = 0$, and (2) if the system should remain in a state, this can be modeled using $ta(s) = +\infty$.

- $\delta_{int} : S \to S$ the *internal transition function*, which models the transition from one state to the next sequential state
- $X = \times_{i=1}^{m} X_i$ the *set of admissible inputs*, formalizing multiple (m) input ports, each identified by a unique index i (possibly derived from a port name)
- $\delta_{ext} : Q \times X \to S$ the *external transition function* describing how the system responds to *external events*, with $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ the *total state set*, where e is the *elapsed time* since transitioning to the current state s
  Note: the *time left* $\sigma = ta(s) - e$ is often used.
- $Y = \times_{i=1}^{l} Y_i$ the *set of admissible outputs*, formalizing multiple (l) output ports, each identified by a unique index i (possibly derived from a port name)
- $\lambda : S \to Y \cup \{\phi\}$ the *output function*, which determines which output events are generated at the time of an *internal transition*.
  Note: (1) the state *before* the transition is used as input to λ, and (2) output events are *only* generated on *internal* transitions and *not* on *external* ones.

As an example, the atomic DEVS representation of the "TrafficLight" model shown in Figure 3 is given below:

$$TrafficLight = \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle$$

$$S = \{green, yellow, red, manual\}$$

$$ta = \begin{Bmatrix} green \to 50\,\text{s}, \\ yellow \to 10\,\text{s}, \\ red \to 60\,\text{s}, \\ manual \to +\infty \end{Bmatrix}$$

$$\delta_{int} = \begin{Bmatrix} green \to yellow, \\ yellow \to red, \\ red \to green \end{Bmatrix}$$

$$X = X_1 = \{toManual, toAuto\}$$

$$\delta_{ext} = \begin{Bmatrix} (green, e),\ toManual \to manual, \\ (yellow, e),\ toManual \to manual, \\ (red, e),\ toManual \to manual, \\ (manual, e),\ toAuto \to red \end{Bmatrix}$$

$$Y = Y_1 = \{toGreen, toYellow, toRed\}$$

$$\lambda = \begin{Bmatrix} green \to toYellow, \\ yellow \to toRed, \\ red \to toGreen \end{Bmatrix}$$

The atomic DEVS representation of the 'Police' model is omitted here for brevity.

Coupled DEVS Models. These models build on atomic models to represent larger systems made up of interconnected components. They describe a system as a network of coupled components (models), with connections between components denoting how they influence each other. More specifically, through a connection, output events of one component can become input events for another (internal coupling). These components can be atomic DEVS models, but also other coupled DEVS models. As such, DEVS allows for a hierarchical modeling approach.[17] Additionally, components can be connected to input/output ports of an encompassing coupled DEVS model (external coupling).

A coupled DEVS model can be defined as follows:

$$coupledDEVS \equiv$$
$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

With:

- $self$ denoting the coupled model itself
- X_{self} the (possibly structured) *set of allowed external inputs* to the coupled model
- Y_{self} the (possibly structured) *set of allowed (external) outputs* of the coupled model
- D a *set of unique component references*, not including *self*
- $\{M_i\}$ the *set of components*, where each component is an atomic DEVS model referenced by D
- $\{I_i\}$ the *set of influencees*, determining the components influenced by a component $i \in D \cup \{self\}$, i.e., the components whose input ports are connected to output ports of component i
  Note: a component (1) can not influence components outside the coupled model, and (2) can not influence itself directly
- $\{Z_{i,j}\}$ the *set of translation functions*, used to translate an output event of one component $i \in D \cup \{self\}$ to a corresponding input event in an influencee $j \in I_i$ of that component (if necessary)
  Note: if not defined, these are typically implicitly assumed to be the identity function.[18]

- *select* the *tie-breaking function* used to resolve collisions between components, i.e., multiple components set to transition at the same time, by selecting exactly one of those components to execute its transition

Figure 3 shows an illustration of such a coupled DEVS model ("TrafficSystem"), consisting of two atomic DEVS models of a police officer and a traffic light. In *autonomous* mode, the traffic light automatically transitions between the states "green", "yellow", and "red". When the police officer starts working (transitions to the "working" state), an output event "toManual" is generated, which causes an external transition in the "TrafficLight" model ("?toManual"), transitioning it to a *manual* mode ("manual"). Similarly, when the police officer stops working, the traffic light is switched back to *autonomous* mode, starting in the "red" state. The coupled DEVS representation of this "TrafficSystem" model is given below:

$$TrafficSystem = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

$$X_{self} = \phi$$

$$Y_{self} = \{toGreen, toYellow, toRed\}$$

$$D = \{light, officer\}$$

$$\{M_i\} = \left\{ \begin{array}{c} M_{light} = TrafficLight, \\ M_{officer} = Police \end{array} \right\}$$

$$\{I_i\} = \left\{ \begin{array}{c} light \rightarrow \{self\}, \\ officer \rightarrow \{light\} \end{array} \right\}$$

$$\{Z_{i,j}\} = \left\{ \begin{array}{c} Z_{light,self} = \left\{ \begin{array}{c} toGreen \rightarrow toGreen, \\ toYellow \rightarrow toYellow, \\ toRed \rightarrow toRed \end{array} \right\} \\ Z_{officer,light} = \left\{ \begin{array}{c} toAuto \rightarrow toAuto, \\ toManual \rightarrow toManual \end{array} \right\} \end{array} \right\}$$

$$select = \left\{ \begin{array}{c} \{light, officer\} \rightarrow officer, \\ \{light\} \rightarrow light, \\ \{officer\} \rightarrow officer, \end{array} \right\}$$

Parallel DEVS (PDEVS)

Parallel DEVS was presented by Chow et al.[2] as a revision of the "classic" DEVS formalism to better support parallelism. Instead of relying on a *select* function to resolve collisions at the coupled model level, it allows the modeler to explicitly define the collision behavior at the atomic model level using a new *confluent transition function* (δ_{conf}). Often, the confluent transition function is defined as first invoking the internal transition function, followed by the external transition function.[19] Additionally, the *external* and *confluent transition functions* now operate on *bags* instead of single inputs. Similarly, the *output function* produces a *bag* instead of a single event.

Therefore, an atomic Parallel DEVS model can be defined as follows:

$$atomicPDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle$$

Note the addition of the *confluent transition function* (δ_{conf}) compared to "classic" DEVS.

And a coupled Parallel DEVS model can be defined as follows:

$$coupledPDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

Note the removal of the *select* function compared to "classic" DEVS.

PDEVS Abstract Simulator. An abstract simulator defines the operational semantics of the formalism. Chow et al.[20] provide the original description of the abstract simulator for Parallel DEVS. A possible implementation of this abstract simulator is given in Algorithms 1 to 3. For a more extensive description, the reader is referred to the original work.

While Chow et al. describe the algorithms in terms of messages being passed between them, highlighting their parallelizability, we instead describe them in terms of (blocking) function calls to more easily relate them to the FMI functions in subsequent discussions.

First, Algorithm 1 presents the simulator for atomic models, while Algorithm 2 presents the orchestrator for coupled models. Note how the functions implemented in the atomic simulator and coupled orchestrator have similar signatures. This allows the simulation to be constructed hierarchically, i.e., components of the coupled model can again be coupled models with their own orchestrators. Algorithm 3 presents the root-level orchestrator to run the simulation by interacting with the topmost coupled model's orchestrator.

The presented algorithms use the following variables:

- t: simulation time
- t_N: simulation time at which the next internal transition will occur
- t_L: simulation time at which the last transition occurred
- s: state of the (atomic) model
- e: simulation time elapsed since the last transition
- x: input event(s)
- y: output event(s)
- y_{ext}: external output event(s)
- $self$: reference to the model itself
- *IMM*: set of model references for models with imminent transitions
- *INF*: set of model references for influenced models, i.e., models with input events

Comparison between FMI SC and (P)DEVS

Previous works[8,13] have highlighted the similarities and differences between the two formalisms. Both FMI SC and (P)DEVS are designed to handle discrete events in simulation, emphasizing event-based synchronization to ensure consistent interactions across components. While FMI SC standardizes co-simulation interfaces to integrate diverse models, (P)DEVS focuses on system representation

and structured state transitions. Event handling in FMI SC relies on input and output clocks activated at predefined intervals or by external triggers, whereas (P)DEVS uses internal and external event transitions to manage system state changes. (P)DEVS inherently supports hierarchical modeling through coupled models, whereas FMI SC enables modular co-simulation by integrating multiple FMUs, though composing FMUs into coupled FMUs requires additional orchestration which has not been done for FMI SC. FMI SC is also optimized for real-time and co-simulation scenarios, coordinating events across FMUs through synchronous clocks. While (P)DEVS is grounded in a strict mathematical formalism, FMI SC lacks formal semantics and instead relies on external orchestration mechanisms.

Algorithm 1: Parallel DEVS atomic model simulator.

Function *getOutput(t)*:
  if $t = t_N$ **then**
    $y \leftarrow \lambda(s)$
    return y
  else
    raise error
  end
end

Function *setInput(x)*:
  $bag \leftarrow bag \cup \{x\}$
end

Function *doTransition(t)*:
  if $t_L \leq t < t_N$ ***and*** $bag \neq \emptyset$ **then**
    // External transition
    $e \leftarrow t - t_L$
    $s \leftarrow \delta_{ext}(s, e, bag)$
    $bag \leftarrow \emptyset$
    $t_L \leftarrow t$
    $t_N \leftarrow t_L + ta(s)$
  else if $t == t_N$ ***and*** $bag == \emptyset$ **then**
    // Internal transition
    $s \leftarrow \delta_{int}(s)$
    $t_L \leftarrow t$
    $t_N \leftarrow t_L + ta(s)$
  else if $t == t_N$ ***and*** $bag \neq \emptyset$ **then**
    // Confluent transition
    $s \leftarrow \delta_{con}(s, bag)$
    $bag \leftarrow \emptyset$
    $t_L \leftarrow t$
    $t_N \leftarrow t_L + ta(s)$
  else if $t > t_N$ ***or*** $t < t_L$ **then**
    raise error
  end
  return t_N
end

Algorithm 2: Parallel DEVS coupled model orchestrator.

Function *getOutput(t)*:
  if $t == t_N$ **then**
    $t_L \leftarrow t$
    $y_{ext} \leftarrow \emptyset$
    // For each component with an imminent internal transition
    foreach $i \in D$ *with* $M_i.t_N == t$ **do**
      $y \leftarrow M_i.getOutput(t)$
      $IMM \leftarrow IMM \cup \{i\}$
      // Communicate events
      foreach $j \in I_i$ **do**
        if $j \neq$ *self* **then**
          // Internal coupling
          $x \leftarrow z_{i,j}(y)$
          $M_j.setInput(x)$
          $INF \leftarrow INF \cup \{j\}$
        else
          // External coupling
          $y_{ext} \leftarrow y_{ext} \cup z_{i,self}(y)$
        end
      end
    end
    return y_{ext}
  else
    raise error
  end
end

Function *setInput(x)*:
  $bag \leftarrow bag \cup \{x\}$
end

Function *doTransition(t)*:
  if $t_L \leq t \leq t_N$ **then**
    // Forward external inputs
    foreach $j \in I_{self}$ ***and*** $x \in bag$ **do**
      $x \leftarrow z_{self,j}(x)$
      $M_j.setInput(x)$
      $INF \leftarrow INF \cup \{j\}$
    end
    bag $\leftarrow \emptyset$
    // Do component transitions
    foreach $i \in IMM \cup INF$ **do**
      $M_i.doTransition(t)$
    end
    // Update times
    $t_L \leftarrow t$
    $t_N \leftarrow$ minimum of components' t_N
    $IMM \leftarrow \emptyset$
    $INF \leftarrow \emptyset$
    return t_N
  else
    raise error
  end
end

Algorithm 3: Parallel DEVS root orchestrator.

```
// Advance time to first event
t ← t_N of topmost_coordinator
while not stoppingcondition() do
    // Get and exchange output events
    y ← topmost_coordinator.getOutput(t)
    // Do transitions
    t_N ← topmost_coordinator.doTransition(t)
    // Advance time to next event
    t ← t_N
end
```

PDEVS Models of the Running Example

In the running example of Figure 1, the whole adaptive cruise control scenario can be modeled as a coupled PDEVS model, with the controller, supervisor, drivers, and both vehicles as component models. The vehicles periodically change their state (discretizing their continuous dynamics), generating output events with their updated position (x), speed (v), and acceleration (a). Both vehicles also receive external events with a wanted acceleration, provided by either the controller (for the ego vehicle) or by a driver model (for the lead vehicle). The rest of the system can be similarly modeled, using periodically generated output events to trigger external transitions in components, e.g., to update a speed setpoint, wanted acceleration, etc.

An overview of the PDEVS implementation of the ACC example is given below. To keep the main text concise, the formal description of the atomic and coupled PDEVS models is provided in Appendix A: Formal Specification of the ACC Models, Definitions 1 to 8. The PythonPDEVS implementation of these models is also available on GitHub: `https://github.com/Cosys-Lab/2025-SIMULATION-DEVS-FMI3.0`

The ACC system can be modeled in PDEVS as a coupled model (Definition 8) composed of the following components:

- **Ego Vehicle (Coupled) - Definition 6**
  Coupled PDEVS model representing the ego vehicle
  - Vehicle (Atomic) - Definition 1
    Implements a simple kinematic vehicle model
  - Generator (Atomic) - Definition 2
    Used to periodically trigger the kinematic vehicle model to update its state
- **Lead Vehicle (Coupled) - Definition 7**
  Coupled PDEVS model representing the lead vehicle
  - Vehicle (Atomic) - Definition 1
    Implements a simple kinematic vehicle model
  - Generator (Atomic) - Definition 2
    Used to periodically trigger the vehicle model to update its state
  - Sine (Atomic) - Definition 3
    Signal generator (sine wave), represents the driver by periodically updating the wanted acceleration for the vehicle model
- **Controller (Atomic) - Definition 4**
  Atomic PDEVS model implementing a basic speed controller (cruise control), periodically updates the (wanted) acceleration of the ego car
- **Supervisor (Atomic) - Definition 5**
  Atomic PDEVS model implementing the "adaptive" aspect of the cruise control model by periodically updating the speed setpoint for the speed controller to either (1) maintain a safe distance to the lead vehicle, or (2) to obey the speed limit
- **Controller Generator (Atomic) - Definition 2**
  Periodically triggers the controller to update its output (cfr. time-triggered clock in Figure 1)
- **Supervisor Generator (Atomic) - Definition 2**
  Periodically triggers the supervisor to update its output (cfr. time-triggered clock in Figure 1)

Note that the ACC system model is constructed hierarchically, i.e., it contains both atomic PDEVS models and other coupled PDEVS models. It's also these components (Ego Vehicle, Lead Vehicle, Controller, Supervisor, Controller Generator, Supervisor Generator) that are exported as FMUs. As such, this allows us to validate our approach for both types of PDEVS models.

Related Work

There has been much work on integrating DEVS with other formalisms, such as FMI, to enable coupling these two formalisms. At the foundational level, we highlight the work of Vangheluwe et al.[21] where the multiple ways to integrate formalisms are presented, including transformation to a single formalism, semantic adaptation, and co-simulation. Examples of formalism transformation are given in the cited paper, and an example semantic adaptation is given by Mustafiz et al.[22] and with a focus on FMI by Gomes et al.[23] We focus on the co-simulation approach, where the two formalisms are kept separate, and the co-simulation is orchestrated by an external tool. We classify related works into three categories: (1) integrating FMI into DEVS (wrapping FMUs as DEVS), (2) integrating DEVS into FMI (wrapping DEVS as FMUs), and (3) applications leveraging DEVS–FMI integration.

In terms of wrapping FMU as DEVS models, we highlight the work of Camus et al.[24] where they present a hybrid co-simulation approach where FMUs are wrapped to integrate them with DEVS in the MECSYCO middleware. A case study on a barrel-filling system demonstrates the framework's capability to manage event synchronization and numerical resolution, ensuring accurate and scalable co-simulation. The work of Quraishi et al.[25] explores co-simulation of hardware described at the Register-Transfer Level (RTL) and software using the FMI standard. A DEVS-FMI interface integrates DEVS-Suite for RTL models and MATLAB for software models, enabling modular co-simulation of hardware-software interactions. The approach is validated using a Network-on-Chip (NoC) system, demonstrating its ability to simulate hybrid systems efficiently and handle disparate simulation environments. Lin[26] introduces a DEVS-FMI adapter for integrating DEVS-Suite and FMUs, enabling co-simulation of discrete and continuous models. Using a four-variable model framework, the paper evaluates a case study of an electric scooter, demonstrating the adapter's ability to synchronize cyber and physical systems. The work highlights challenges, such as step size effects, and proposes a hybrid co-simulation

strategy for CPS design and validation. In addition, the paper by Joshi et al.[27] introduces a method to integrate Functional Mock-up Units (FMUs) into the DEVS-based Cadmium simulator by wrapping FMUs as DEVS atomic models. The integration leverages a Quantized State System (QSS) solver to simulate continuous-time behavior efficiently, replacing traditional time discretization with state quantization.

With a focus on integrating DEVS models within FMI, we previously introduced a co-simulation framework for evaluating multicore embedded platforms in cyber-physical systems.[28] This previous approach integrates DEVS models of multicore platforms with plant and application models using the FMI standard to enable early design-stage evaluations of temporal and functional behavior. However, this previous approach made use of version 2.0 of FMI, which lacked support for discrete event simulation. To work around these limitations, the previous approach required the DEVS models to be included as part of the FMI orchestration algorithm itself, rather than as FMUs. Therefore, the orchestration algorithm was not generic. More recently, we presented an approach using FMI 3.0. This approach makes use of, among others, the event mode and synchronous clocks introduced in FMI 3.0 to wrap DEVS models in FMUs and to co-simulate them with plant and application models. However, this previous work did not explicitly support the co-simulation of multiple DEVS FMUs. The feasibility of coupling multiple DEVS models through FMI without breaking the DEVS formalism remained an open question, now answered in the current manuscript.

For applications, we highlight Paris et al.[29] where the authors propose a component-based approach to DEVS for enhancing reuse and integration in complex system modeling and simulation. The authors draw parallels with FMI's success for equation-based models and advocate for a DEVS-based standard to facilitate modularity and co-simulation. Using the MECSYCO middleware as an example, the paper illustrates how DEVS can bridge heterogeneous simulation tools, enabling multi-paradigm modeling and co-simulation.

Compared to previous works, we address the limitation of the approach presented in Vanommeslaeghe et al.[8] to demonstrate that FMI 3.0 FMUs can faithfully reproduce PDEVS semantics, and that clocked partitions enable the transport of event data in FMI 3.0, ensuring that coupling PDEVS models through FMI does not violate the PDEVS formalism. In this regard, we are the first to relate FMI SC to PDEVS. Key considerations to adapt PDEVS models for FMI 3.0's interface include the development of mechanisms for signaling events, managing input/output communications, and synchronizing clocks. These are detailed in the next section.

Contribution

To enable the integration of PDEVS models in FMUs and their coupling through FMI, we need to make semantic adaptations between PDEVS and FMI. These adaptations require addressing several key aspects, including handling internal transitions, communicating input and output events, and ensuring that PDEVS models encapsulated in different FMUs can be coupled without violating the PDEVS formalism.

First, in Section Requirements, we define the general requirements that *a* simulator must satisfy to correctly simulate PPDEVS models. These requirements establish the foundation for ensuring accurate and consistent behavior when integrating PDEVS models within FMI and serve as the basis for our approach.

Then, we detail our approach to adapting the semantics of PDEVS to FMI. Specifically, we focus on three main aspects. We describe how the FMI importer can be made aware of internal transitions in PDEVS models, how input and output events can be communicated using the standard FMI interface, and how PDEVS models encapsulated in separate FMUs can be coupled while preserving the PDEVS formalism.

The following subsections elaborate on these aspects, and provide the motivations and considerations behind our chosen approach and its relation to the predefined requirements. Sections Signaling Imminent Internal Transitions and Communicating Input/Output Events provide a static view of the first two aspects, while Coupling DEVS FMUs presents a dynamic view of the last.

Requirements

Based on the abstract simulator (Algorithms 1 to 3), we identify requirements that a simulator must meet — including the overall flow it must follow — to correctly simulate PDEVS models. The simulator:

R0: Must *advance the simulation time* t *to the next event time* t_N, meaning it:

- **R0.a:** Must *advance the simulation time* t
- **R0.b:** Must *obtain the next event time* t_N based on the components' t_N

R1: Must generate outputs y for each component with an imminent internal transition (each model $m \in \{M_i\}$ with $m.t_N == t$), meaning it:

- **R1.a:** Must know *when* the next internal transition should happen for each component ($m.t_N$)
- **R1.b:** Must trigger a call to the *getOutput(...)* function of a component to generate (bags of) outputs y for said component

R2: Must communicate events between component models, based on their coupling ($\{I_i\}$), meaning it:

- **R2.a:** Must get (bags of) outputs y from a component
- **R2.b:** Must set (bags of) inputs x for a component
- **R2.c:** Must respect the coupling defined in $\{I_i\}$

R3: Must tell components to execute their transitions, triggering a call to the *doTransition(...) function*

R4: Must preserve the overall flow imposed by the abstract simulator, specifically:

- **R4.a:** The *getOuput(...)* function must be called before the outputs y are collected

R4.b: Outputs y must be collected before they are communicated, i.e., before inputs x are set

R4.c: All events must be communicated before components' transitions (*doTransition(...)*) are executed to ensure the correct transition function (internal, external, or confluent) is executed

Regarding the mapping to FMI, these requirements determine (1) the overall flow an FMI orchestration algorithm must follow, and (2) the functions that need to be implemented in PDEVS FMUs, to ensure a correct coupling of PDEVS models through FMI. Overall, the FMI orchestration algorithm must impose the same flow as Algorithms 2 and 3 to ensure it satisfies requirement **R4**, while the FMUs must expose the functionality of Algorithm 1 (or Algorithm 2 if they contain a coupled model) through standard FMI functions to enable this.

Signaling Imminent Internal Transitions

To ensure proper event handling, an FMI importer must transition the relevant FMUs into event mode when necessary. This requires the importer to be aware of, and keep track of, imminent events. In the context of PDEVS models, these imminent events correspond to imminent internal transitions. As such, this aligns with requirements **R1.a** and by extension **R0.b**.

To signal these imminent internal transitions, we use a subtype of FMI 3.0 time-based clock called the *countdown aperiodic clock*. More concretely, for our PDEVS FMUs, we define a countdown aperiodic input clock "ta" ("time advance"), whose period is equal to the time until the next internal transition (σ). Therefore, a call to $fmi3GetInterval$ for this clock allows the importer to query a PDEVS FMU for the time until the next internal transition, thereby obtaining the next event time t_N and satisfying **R1.a**. As the FMI importer is responsible for keeping track of when any input clocks *should* tick, this also allows it to satisfy **R0.b**. Listing 1, Line 2 shows how this "ta" clock is defined in an excerpt from a *modelDescription.xml* file from a PDEVS FMU.

Additionally, the importer is responsible for instructing the relevant clocks *to* tick at the right point(s) in time. As later described in Section Implementation Outline, triggering a call to *getOutput(...)* on activation of this "ta" clock also allows us to satisfy **R1.b**.

Rationale. An alternative mechanism provided in FMI 3.0 that could be used to signal imminent internal transitions is the ability for an FMU to signal the early return from a *fmi3DoStep* call. When the importer calls the *fmi3DoStep* function, we could check if the requested step size is greater than the remaining time until the next internal transition in the PDEVS model (σ). If this is the case, we could advance the internal time up to the point of the internal transition, and signal the early return and need for event handling to the importer using the relevant return values.

However, there are some major drawbacks to that approach. First, the importer needs to explicitly support early return. As such, relying on this mechanism would likely limit in which tools these PDEVS FMUs could be used. Second, in the case the importer does support early return, an FMU

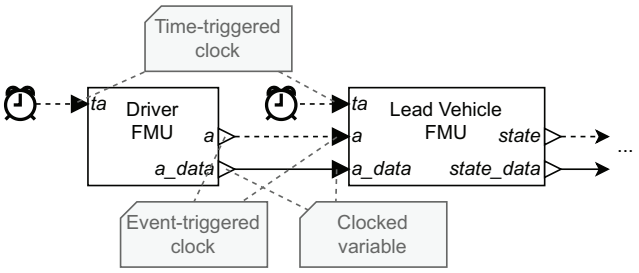

Figure 4. Conceptual illustration of the open loop variant represented using DEVS FMUs.

signaling such an early return might mean that the importer needs to roll back the state of other FMUs which did fully advance their internal time, such that they may be advanced instead to the *lastSuccessfulTime* of the signaling FMU. In addition, this is an optional feature that needs to be explicitly supported by the FMUs. In our experience, most FMUs do not support this feature, as it is not mandatory. As such, this would likely severely limit the co-simulation setups in which these PDEVS FMUs could be used. Additionally, relying on this mechanism makes it difficult to satisfy requirements **R0.b** and **R1.a**, as the importer does not know a priori when an event will occur.

To avoid the mentioned drawbacks of relying on the early return mechanism, we opt for the *countdown aperiodic clock* instead. As it is basic required functionality for an FMI 3.0 compatible importer (1) to keep track of when input clocks *should* tick, and (2) to move the FMUs into event mode and to instruct the relevant clocks *to* tick at the right point(s) in time, we believe that relying on this mechanism for the PDEVS FMUs makes it more likely for them to be usable in different (co-)simulation tools and scenarios.

Communicating Input/Output Events

As mentioned in Section Discrete Event System Specification (DEVS), (P)DEVS models can generate output events on internal transitions, and can transition between states in response to input events (external transitions). To enable the coupling of PDEVS FMUs through FMI, we need to be able to communicate the occurrence of these events using some interface defined in the FMI standard (**R2**).

Version 3.0 of the FMI standard introduces *clocked variables*, which are discrete variables explicitly tied to the activation of one or more *clocks*. These variables are subject to access restrictions to ensure synchronization with their associated clocks. Specifically, clocked variables can only be accessed when one of the referenced clocks is active. This ensures that the variables are updated and accessed in sync with their associated clocks.

In the current work, we adopt this concept of clocks and clocked variables to model input/output ports of PDEVS models. For each input/output *port* of a PDEVS model, we define both a triggered input/output *clock* and an associated *clocked variable*. Conceptually, the *clocks* indicate the *occurrence* of events on a specific port, while the *clocked variables* are used to carry the associated *data* (i.e, the actual event) for each port, updated strictly when their corresponding clock signals an event. As further detailed in Section Implementation Outline, these clocks and clocked

```
1 <ModelVariables>
2   <Float64 name="time" valueReference="999"
        causality="independent" variability="
        continuous" description="Simulation time"/
        >
3   <Clock name="ta" valueReference="1001"
        causality="input" intervalVariability="
        countdown"/>
4   <String name="state" valueReference="1"
        causality="output" variability="discrete"/
        >
5   <Clock name="update_a_wanted" valueReference="
        1002" causality="input"
        intervalVariability="triggered"/>
6   <String name="update_a_wanted_data"
        valueReference="2" causality="input"
        variability="discrete" clocks="1002">
7     <Dimension start="1"/>
8     <Start value=""/>
9   </String>
10  <Clock name="vehicle_state" valueReference="
        1003" causality="output" clocks="1001"
        intervalVariability="triggered"/>
11  <String name="vehicle_state_data"
        valueReference="3" causality="output"
        variability="discrete" clocks="1003"/>
12 </ModelVariables>
13 <ModelStructure>
14   <Output valueReference="1003" dependencies="
        1001"/>
15 </ModelStructure>
```

Listing 1: Definition of the aperiodic countdown clock and input/output clocks with associated clocked variables for the lead vehicle model.

variables allow the FMI importer to get outputs and set inputs using standard FMI functions, satisfying **R2.a** and **R2.b** respectively.

Additionally, as outputs may be generated on imminent internal transitions, as signaled using the "ta" *clock*, we define a dependency between each *triggered output clock* and the "ta" *clock*. This serves to satisfy **R4.a**.

An example of this can be seen in Listing 1, which shows an excerpt of the model description for the ego vehicle model. The PDEVS model of the ego vehicle (Definition 6) has an input port "update_a_wanted" and an output port "vehicle_state". As such, the listing shows a triggered input clock "update_a_wanted" (line 3), and a triggered output clock "vehicle_state" (line 8). It also shows two clocked variables, "update_a_wanted_data" and "vehicle_state_data" (lines 4, 9), with a "clocks" attribute linking them to their associated clocks. These values are computed and queried while their corresponding clocks are active. (Recall Section Orchestration Algorithm for FMI 3.0 Synchronous Clocks.) Additionally, Line 14 shows the dependency between output clock "vehicle_state", and the "ta" clock.

Note that in our current implementation of the presented approach, events are serialized to *base64* encoded strings to provide a generic implementation that is independent of the types of events being exchanged. Hence, the shown clocked variables are of type "String". However, other data types might be used depending on the specific implementation.

Rationale. This approach addresses limitations of our previous work,[8] where we relied solely on triggered input/output *clocks* to communicate events. More explicitly, in that earlier approach, we defined a triggered input clock for each admissible input *event* in X and a triggered output clock for each admissible output *event* in Y (rather than for each *port*). Then, we defined mappings associating each input/output clock (using their value reference) to specific events in X and Y respectively. While this approach was shown to be sufficient for the use cases presented in the previous paper, including the traffic system example also shown in Figure 3, it does not scale to more complex (P)DEVS models. Defining input/output clocks for each $x \in X$ and $y \in Y$ is feasible only when the sets are small. However, when events contain data, e.g., the wanted acceleration output by the controller in the adaptive cruise control example, this previous approach becomes unusable. As such, in the current work, we propose an alternative approach using both *clocks* and *clocked variables*.

Coupling DEVS FMUs

In previous work,[8] we showed how we could use a single DEVS FMU to trigger other (non-DEVS) FMUs to simulate the behavior of a cyber-physical system. However, one major question that remained unanswered was whether multiple DEVS FMUs could be co-simulated (coupled) using FMI without violating the DEVS formalism. In this section, we address this question.

First, we describe the difficulties of coupling "classic" DEVS models, as used in our previous work,[8] using FMI, and motivate a switch to Parallel DEVS in the current work. Then, we outline an approach to integrate PDEVS in FMUs which satisfies the requirements defined in Section Requirements. Lastly, we present an example orchestration algorithm, based on one proposed by Ravi et al.,[16] that further satisfied these requirements.

Challenges in Coupling Classic DEVS Models. One major point of attention in coupling DEVS models in general is that due to the coupling of multiple concurrent components (models), multiple state transitions can happen at the same point in time. In sequential simulation systems, such transition collisions are resolved by means of some form of selection of which of the components' transitions should be handled first. This corresponds to the introduction of priorities in some simulation languages.[17]

In classic DEVS, coupled models explicitly include a *select* function for tie-breaking between simultaneous transitions. This function is used by the DEVS solver to select one component from a set of components with simultaneous transitions, essentially prioritizing certain components over others and serializing the solving of the coupled model. As such, to couple DEVS models through FMI, *we must be able to similarly prioritize the different FMUs containing the component DEVS models* to ensure the FMI co-simulation exhibits the same behavior as a pure DEVS simulation. However, as this *select* function is defined at the level of the coupled model, and the coupling is to be performed by the FMI importer, this complicates the implementation as the importer must somehow preserve the functionality of the *select* function.

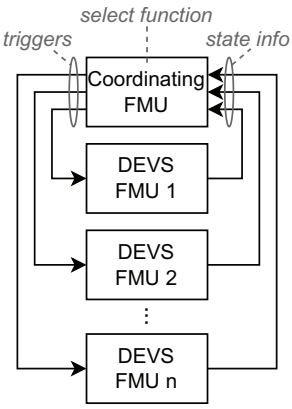

Figure 5. Illustration of the coordinating FMU concept for "classic" DEVS.

Given these challenges, a more effective approach is to adopt Parallel DEVS. PDEVS eliminates the need for a *select* function by introducing explicit mechanisms for handling concurrent transitions. Instead of requiring an external entity to impose a priority order, PDEVS allows components to execute transitions in parallel while ensuring deterministic behavior through well-defined confluent transition functions. As these confluent transitions are defined at the atomic model level, this significantly simplifies the integration of PDEVS model in FMUs and their coupling through FMI while preserving the intended behavior of the PDEVS formalism.

Rationale. As mentioned, in "classic" DEVS, coupled models incorporate a *select* function to resolve simultaneous transitions. This function determines which component takes priority when multiple transitions occur at the same time, effectively serializing the solving of the coupled model. Therefore, to correctly couple DEVS models through FMI, the simulation must allow for similar prioritization of FMUs containing individual DEVS components. This to ensure that the co-simulation behaves consistently with a pure DEVS simulation.

For simple cases in which there are no loops, such as the traffic system (Figure 3), or indeed the open loop scenario in Figure 1, it *might* be possible to ensure a specific ordering of FMUs by strategically defining dependencies between different input/output clocks or clocked variables. However, this assumes (1) that the priorities match the flow of data/events and (2) that the priorities do not change. As such, this approach would be severely limited compared to the expressiveness offered by the *select* function.

In any case, for more complex cases, such as the closed loop scenario in Figure 1, a different approach is needed. One option here would be to include the select function, or a version thereof, as part of the orchestration itself. However, this approach has drawbacks. First, it results in a DEVS-specific orchestration algorithm. Not only that, as the select function can be specific to a given coupling, this can even result in a simulation-specific orchestration algorithm. As such, such an approach would severely limit the possibility of using DEVS FMUs in different co-simulation setups.

Another option could be to wrap the select function itself as a separate "coordinating" FMU, which could then coordinate the component DEVS FMUs using triggered clocks. However, this approach also has its drawbacks. The coordinating FMU would generally require certain information from the different DEVS FMUs, including their current state, time remaining, etc. This concept is illustrated in Figure 5. This information would need to be communicated using FMI, which would complicate the co-simulation setup. Moreover, as the select function can be specific to a given coupling, modifications to the coupled model or the select function would likely require the coordinating FMU to be re-generated and could require extensive modifications to the co-simulation setup (connections), e.g., when different component models are added or removed.

To avoid these drawbacks, we instead propose to switch to Parallel DEVS over classic DEVS. Parallel DEVS does away with the select function and introduces other changes to the DEVS formalism to explicitly handle the occurrence of concurrent events/transitions.

Implementation Outline. The following outlines an implementation of "PDEVS FMUs" and their coupling using standard FMI 3.0 functions satisfying the requirements defined in Section Requirements, with relevant parts also represented in a sequence diagram (Figure 6):

R0: *Advance the simulation time to the next event time*

- **R0.a:** *Advance the simulation time*
  Importer: use *fmi3DoStep(...)* to tell FMUs to advance in time
  FMU: on a call to *fmi3DoStep(...)*: advance the local simulation time, as well as the *elapsed time (e)* in the **PDEVS model**
- **R0.b:** *Obtain the next event time*
  Importer: keep track of the time when "ta" clocks are supposed to tick, selection of the next event time is handled in the **orchestration algorithm**, as detailed in Section Orchestration Algorithm

R1: *Generate outputs for each component with an imminent internal transition* (See also Section Signaling Imminent Internal Transitions)

- **R1.a:** *Know when internal transitions should happen*
  Importer: use *fmi3GetIntervalDecimal(...)* for the *ta* clock to find when it should tick next
  FMU: on a call to *fmi3GetIntervalDecimal(...)* for the *"ta" clock*, return the *time remaining* (σ) from the **PDEVS model**
- **R1.b:** *Trigger a call to the getOutput(...) function*
  Importer: use *fmi3SetClock(...)* to activate the *"ta" clock*
  FMU: on a call to *fmi3SetClock(...)* to activate the *ta* clock, call the *getOutput(...)* function for the **PDEVS model** and store outputs y

R2: *Communicate events between models* (See also Section Communicating Input/Output Events)

- **R2.a:** *Get (bags of) output(s)*
  Importer: use *fmi3GetClock(...)* to get the activation state of triggered output clocks
  FMU: on a call to *fmi3GetClock(...)*, return

fmi3ClockActive if there are stored events for the associated output port of the **PDEVS model**, else return *fmi3ClockInactive*
Importer: if *fmi3GetClock(...)* returned *fmi3ClockActive*, use *fmi3GetString(...)* on the associated clocked variable to get (serialized) events (this information is in the model description)
FMU: on a call to *fmi3GetString(...)*, return the (serialized) stored events for the associated output port of the **PDEVS model**

R2.b: *Set (bags of) input(s)*
Importer: use *fmi3SetClock(...)* to activate triggered input clocks connected to an active output clock (from R2.a)
FMU: on a call to *fmi3SetClock(...)*, store the activation state of the clock
Importer: use *fmi3SetString(...)* on the associated clocked variable to set the (serialized) events obtained from the connected output port (from R2.a)
FMU: on a call to *fmi3SetString(...)*, deserialize the events for the associated input port of the **PDEVS model** and store them, cfr. *bag* in *setInput(...)*

R2.c: *Respect the coupling*
Importer: connections between FMUs are defined using *external relations*, these are used by the **orchestration algorithm** when communicating outputs to inputs

R3: *Tell models to execute their transitions*
Importer: use *fmi3UpdateDiscreteStates(...)* to signal a converged solution at the current super-dense time instant (this function must be called at least once per super-dense time instant as per the FMI standard)
FMU: on a call to *fmi3UpdateDiscreteStates(...)*, call *doTransition(...)* to execute the relevant state transition function(s) in the **PDEVS model**

R4: *Preserve the overall flow*
Importer: the **orchestration algorithm** satisfies this requirement and its subrequirements, as detailed in Section Orchestration Algorithm

Note: (1) the **PDEVS model** inside the **FMU** as mentioned can be either an atomic or coupled model, and (2) certain checks, e.g., checking that a clocked variable being set has at least one active associated clock, have been omitted from the steps above for clarity.

Orchestration Algorithm. Algorithm 4 shows an example algorithm to orchestrate the co-simulation of DEVS FMUs. Note that while this orchestration algorithm is somewhat simplified for this use case, i.e., it contains the bare minimum handling for the continuous-time part, it contains no special "PDEVS-specific" constructs, i.e., the discrete-event handling is as generic as possible. Indeed, our orchestration algorithm is based on the one presented by Ravi et al.[16] and overall matches their version closely.

The relations of different parts of the orchestration algorithm to the requirements are mostly indicated using comments in the pseudocode. Some important parts not fully covered in the outline (Section Implementation Outline) are as follows:

- **R0.b:** *Must be able to obtain the next event time t_N based on the components' t_N*
  The importer keeps track of when time-based clocks, such as the "ta" clocks, need to tick. The selection of the next event time is handled by the orchestration algorithm using the function *getEarliestTickTime()* as shown in Algorithm 4, Lines 3 and 57. This function selects the earliest time at which one of the time-based clocks ("ta" clocks) needs to tick, i.e., when the next internal transition will occur. Hence, it serves to satisfy **R0.b**.
- **R2.c:** *Must respect the coupling defined in $\{I_i\}$*
  As previously mentioned, connections between FMUs are defined using external relations, which are used by the orchestration algorithm when communicating outputs to inputs. This can be seen in the orchestration algorithm on Lines 32, where *getConnectedOutput(...)* resolves the external coupling for a specific input port to a specific output port. Then, on Line 33, this is used to set the input value based on the connected output value. Additionally, on Lines 44-46, the external relations are used to communicate the occurrence of events from output clocks to connected input clocks. However, one limitation of the FMI standard in this regard is that external relations can only be "one-to-many", while PDEVS supports a "many-to-many" coupling. As such, these external relations only partly satisfy **R2.c**.

In general, the orchestration algorithm must preserve the overall flow imposed by the abstract simulator (**R4**). More specifically:

- **R4.a:** *The getOuput(...) function must be called before the outputs y are collected*
  As described in Section Implementation Outline, a call to activate the "ta" clock triggers a call to the *getOutput(...)* function in the FMU. This can be seen in Algorithm 4, Lines 25-28. Additionally, as described in Section Signaling Imminent Internal Transitions, we define a dependency between output clocks and the "ta" clock. This dependency is reflected in the orchestration as an *internal relation* between the "ta" clock and triggered output clocks associated with outputs. In the orchestration algorithm, Lines 35-38 use these internal relations to determine which ports have output events. As such, these dependencies/internal relations serve to satisfy **R4.a**.
- **R4.b:** *Outputs y must be collected before they are communicated, i.e., before inputs x are set*
  This requirement is satisfied by the *external relations* between output clocks and input clocks, as well as the dependencies between clocks and clocked variables as described in Section Communicating Input/Output Events. On one iteration of the event handling, the orchestration algorithm uses the output clocks to determine the occurrence of events (Lines 35-38). Then, for active output clocks, it stores the value of the associated clocked output variables (Lines 41-43) and flags connected input clocks for activation (Lines

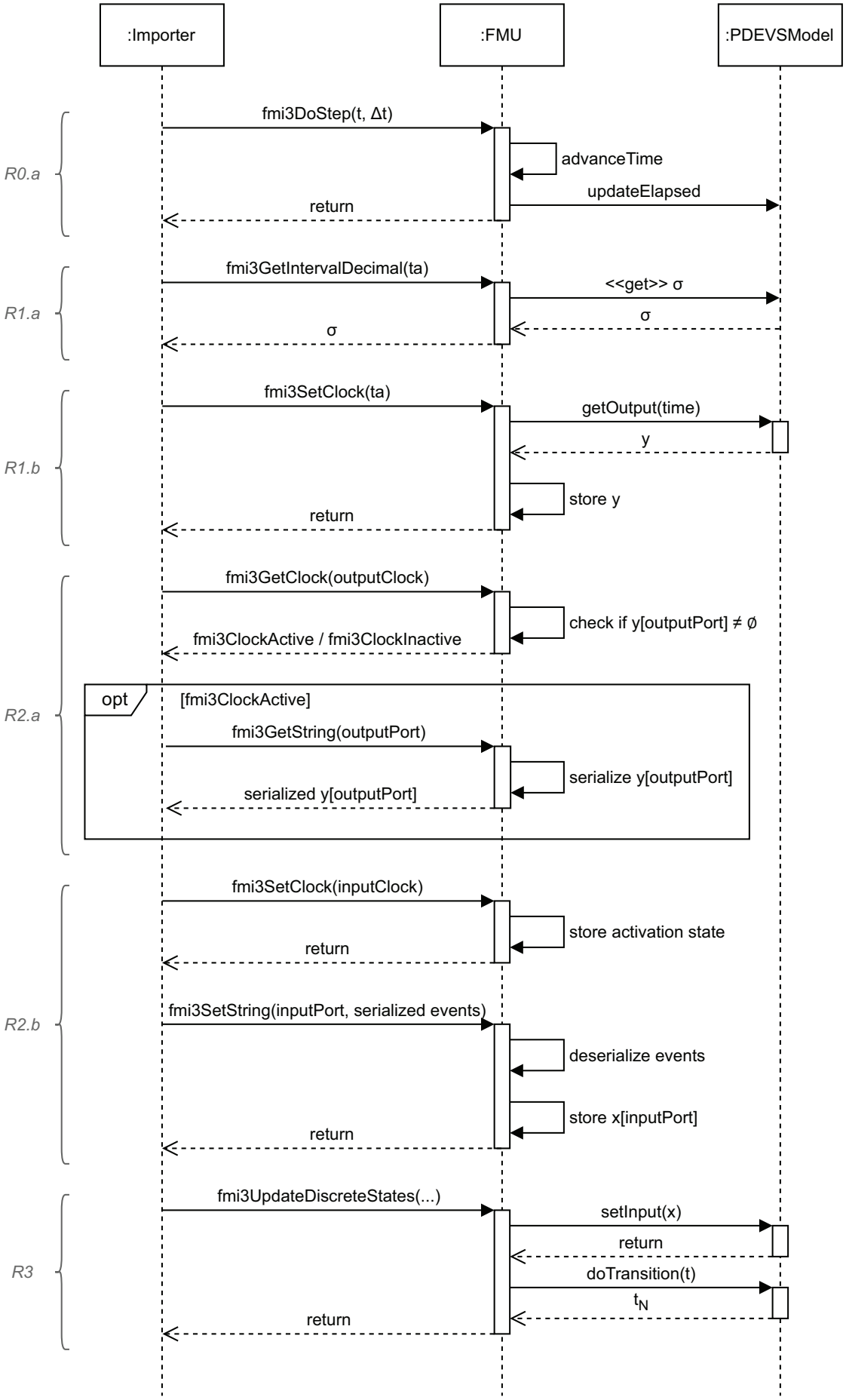

Figure 6. Sequence diagram of the Implementation Outline.

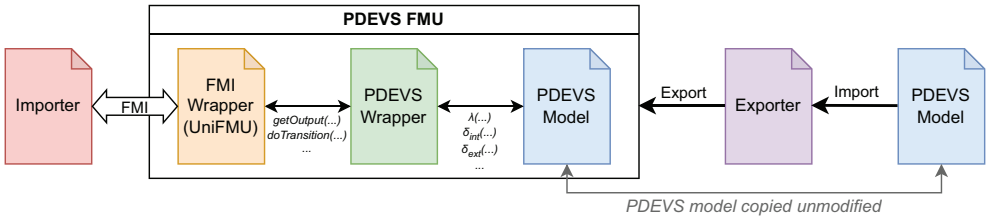

Figure 7. Overview of the implementation.

44-46). Then, in the next iteration, connected input clocks are activated (Lines 25-28) and their associated clocked input variables are set (Lines 31-34). As such, this ensures output are collected before connected inputs are set, satisfying **R4.b**.

- **R4.c:** *All events must be communicated before components' transitions (doTransition(...)) are executed...* As described in Section Implementation Outline, we use the function *fmi3UpdateDiscreteStates(...)* to trigger a call to the *doTransition(...)* function. As per the FMI 3.0 standard, this function is called to signal a *converged solution* at the current super-dense time instant. Therefore, this function gets called when there are no more clocks needing activation, i.e., no more events to communicate, as can be seen on Lines 50-52. As such, this satisfies requirement **R4.c**.

Results and Discussion

A prototype implementation was made to validate the presented approach using the adaptive cruise control use case presented in Section Running Example. First, Section Implementation details our implementation. After this, in Validation, we present the results of our validation efforts. Our prototype implementation and all code required to reproduce the presented results is available on GitHub: `https://github.com/Cosys-Lab/2025-SIMULATION-DEVS-FMI3.0`

Implementation

Figure 7 shows an overview of our implementation. The different parts are detailed below. From left to right:

Importer. We implemented an example FMI importer to demonstrate the presented approach. It provides functionality for importing FMUs and defining external relationships (connections) between ports of those FMUs, provides logging capabilities, and implements the presented Orchestration Algorithm (Algorithm 4) to run the co-simulation. Hence, it implements the "**Importer**" part of the steps outlined in paragraph Implementation Outline. The importer is implemented in Python, making use of the FMPy library [30] to interact with the FMUs.

PDEVS FMU. The PDEVS FMUs were largely implemented in Python by using UniFMU [31] and its Python backend. UniFMU is a tool that facilitates the implementation of FMI-compatible FMUs in programming languages that cannot (easily) produce C-compatible binaries. It achieves this by providing generic binaries that implement the standard FMI functions. These binaries can forward the FMI function-calls to several supported language-specific backends, where the user can implement their required functionality. Overall, the FMUs consist of three main parts: an FMI wrapper, a PDEVS wrapper, and the PDEVS model itself.

FMI Wrapper. The FMI wrapper is implemented in the UniFMU Python backend. It provides the translation between the standard FMI functions (*fmi3...(...)*) and the functions used to interact with the PDEVS models (*getOutput(...)*, *setInput(...)*, *doTransition(...)*), including the serialization and deserialization of events. Therefore, it implements the "**FMU**" part of the steps outlined in paragraph Implementation Outline.

PDEVS Wrapper. The PDEVS wrapper (Python) implements the functions used to interact with the PDEVS models. Depending on the type of the PDEVS model being wrapper, i.e., atomic or coupled, it acts as respectively a simulator (Algorithm 1) or orchestrator (Algorithm 2). This enables a more generic implementation of the PFMI wrapper that can handle both atomic and coupled models.

In the case of a coupled model being wrapped, the model hierarchy is first flattened using the direct connection technique.[32] This technique transforms any hierarchically constructed coupled model into an equivalent flat coupled model with only directly connected atomic models, without changing the behavior of the coupled model. This is not strictly necessary, but simplifies the implementation as it negates the need to implement a full hierarchical simulator in the PDEVS wrapper. In our implementation, this step is carried out using the implementation of this technique provided by PythonPDEVS.[10]

PDEVS Model. This is the PDEVS model being wrapped. As such, this constitutes the "**PDEVS model**" part of the steps outlined in paragraph Implementation Outline. The PDEVS models are implemented in Python using PythonPDEVS [10] and are included in the FMU without modification.

Exporter. To automate the process of creating the FMUs from PDEVS models, we implemented a prototype exporter. The exporter is implemented in Python and allows PythonPDEVS models to be exported to FMUs using a single function-call. Additionally, the exporter provides functionality to easily set up a co-simulation equivalent of a coupled model. Meaning, from a coupled model, the exporter can generate a folder containing an FMU for each component of that model, together with a JSON file describing the coupling of those components.

PDEVS Model. This is the PDEVS model being wrapped, and is the same as the *PDEVS Model* part of Section PDEVS FMU. The models are implemented using a standard (unmodified) version of PythonPDEVS, and needs no

Algorithm 4: Example FMI orchestration algorithm.

```
Function runUntil(stop_time):
    time ← 0
    t_next ← getEarliestTickTime()                                              // R0.b
    while t_next ≤ stop_time do
        step_size ← t_next − time
        foreach fmu ∈ fmus do                                                   // R0.a
            fmu.enterStepMode()
            fmu.doStep(time, step_size, True)
        end
        time ← t_next
        foreach fmu ∈ fmus do
            fmu.enterEventMode()
        end
        it ← 0
        eventHandlingNeeded ← True
        while eventHandlingNeeded do
            clocksNeedingActivation ← {clock ∈ time_based_clocks | clock.nextTickTime = time}   // R1
            if clocksNeedingActivation == ∅ then
                eventHandlingNeeded ← False
                break
            end
            while clocksNeedingActivation ≠ ∅ do
                activeInputClocks ← ∅
                activeOutputClocks ← ∅
                foreach clock ∈ clocksNeedingActivation do
                    clock.activateClock()                  // R1.b for ``ta'', R2.b for inputs
                    activeInputClocks ← activeInputClocks ∪ clock
                end
                clocksNeedingActivation ← ∅
                foreach inClock ∈ activeInputClocks do
                    foreach inPort ∈ dataPorts where inClock ∈ inPort.clocks do   // clocked inputs
                        outPort ← getConnectedOutput(inPort)                    // R2.c
                        inPort.setValue(outPort.value)                          // R2.b
                    end
                    foreach outClock ∈ inClock.internalRelations where
                          outClock.getState() == active do          // activated output clocks, R4.a
                        activeOutputClocks ← activeOutputClocks ∪ outClock       // R2.a
                    end
                end
                foreach outClock ∈ activeOutputClocks do
                    foreach outPort ∈ dataPorts where outClock ∈ outPort.clocks do   // clocked outputs
                        outPort.value ← outPort.getValue()                      // R2.a
                    end
                    foreach inClock ∈ outClock.externalRelations do  // connected input clocks, R2.c
                        clocksNeedingActivation.append(inClock)
                    end
                end
                it ← it + 1
            end
            foreach fmu in fmus do
                fmu.updateDiscreteStates()                                  // R3, R4.c
            end
            foreach clock in time_based_clocks do
                clock.updateNextTickTime(time)                              // R1.a
            end
        end
        t_next ← getEarliestTickTime()                                      // R0.b
    end
end
```

Table 1. Analysis of the differences in values communicated in events between our approach and PythonPDEVS.

Component	Value	Max. Abs.	Mean	Std.
Overall	Time	6.39×10^{-13}	2.13×10^{-13}	2.28×10^{-13}
Ego Vehicle	Position	1.50×10^{-11}	4.32×10^{-12}	6.24×10^{-12}
	Speed	7.85×10^{-13}	-7.65×10^{-14}	1.49×10^{-13}
	Acceleration	2.15×10^{-12}	4.71×10^{-15}	4.19×10^{-13}
Lead Vehicle	Position	1.50×10^{-11}	4.19×10^{-12}	6.28×10^{-12}
	Speed	4.51×10^{-13}	-1.05×10^{-13}	1.18×10^{-13}
	Acceleration	4.13×10^{-15}	5.61×10^{-17}	1.15×10^{-15}
Controller	Wanted Acceleration	1.01×10^{-11}	4.67×10^{-15}	1.52×10^{-12}
Supervisor	Wanted Speed	8.53×10^{-13}	-7.60×10^{-14}	1.61×10^{-13}

modifications or specific construct to be compatible with the presented approach. For all intent and purposes, they are "normal" PythonPDEVS models.

Validation

Validation Strategy. To validate our approach, we use our prototype exporter to generate FMUs for the different components of the ACC system model (i.e., Ego Vehicle, Lead Vehicle, Controller, Supervisor, Controller Generator, Supervisor Generator) described in PDEVS Models of the Running Example. We then use our example importer to set up and run a co-simulation using the generated PDEVS FMUs. We then compare the co-simulation with the PythonPDEVS simulation in two ways.

First, we *numerically* compare the communicated events where possible (i.e., vehicle states, controller output, supervisor output), as well as the simulation time at which events occur. The reasoning being that if our co-simulation approach is correct, the obtained simulation traces should be the same as those obtained using PythonPDEVS.

Second, we directly compare the *behavior* of the PDEVS models themselves. Meaning, which transitions occur at what point in simulation time, with which inputs, which outputs are generated, etc. We do this by instrumenting each atomic PDEVS model, having them generate a log of each state transition. We then compare the logs for each atomic PDEVS model between the co-simulation and the PythonPDEVS simulation to ensure they are equivalent. Logs are deemed equivalent if all the same transitions occur in the same order, with the same input and/or output events on the same ports, at the same time instant. The reasoning being that if our co-simulation approach is correct, the logs should be equivalent to those obtained using only PythonPDEVS.

Rationale. The reason for comparing behavior (logs) at the level of atomic models is that events in coupled models can occur in parallel. However, these parallel events are recorded sequentially in the logs. This means that logs for coupled models can have multiple valid orderings. As a result, comparing logs at the level of coupled models becomes more complex because differences in ordering may not necessarily indicate issues in the co-simulation. The behavior of atomic models, however, is inherently sequential. Therefore, our reasoning is that issues with the orchestration will likely be visible as differences in the logs.

Discussion

First, the results of comparing the values of the communicated events are shown in Table 1. This table shows an analysis of the differences between the (FMI) co-simulation and the PythonPDEVS simulation. For each relevant component of the ACC system model, it shows the mean absolute error, and the mean and standard deviation of the error for different values in the communicated events (for 801 samples each). Additionally, it shows this same analysis for the difference in the time at which events occur in both simulations. From the results, it's clear that the simulation results do not match exactly. However, the differences between the two simulations are small, with the biggest differences being on the order of 10^{-11}. These numerical differences appear to be caused by issues related to floating-point representation and/or conversions occurring somewhere in our implementation, which involves both compiled (C) binaries (UniFMU) and Python code. Examining the obtained traces, we observe small discrepancies in numerical values for our approach compared to PythonPDEVS, which is fully implemented in Python. For example, we observe timestamps with a value of 0.7999999999999999 instead of the expected 0.8. As several of our PDEVS models make use of the *elapsed time* when updating their internal state, such as the new position and speed of a vehicle, it stands to reason that these differences in time cause small differences in the states, which cause further differences in the closed-loop behavior of the simulation.

Second, we use the logs to directly compare the behavior of each atomic PDEVS model, as described in Section Validation Strategy. However, We need to take into account the previously observed numerical differences when comparing the logs. As such, we add an absolute tolerance of 10^{-10} when comparing timestamps and input/output events between the two approaches. Excerpts from the logs for the ego vehicle model when using PythonPDEVS and our approach are shown in Listings 2 and 3 respectively.

In both excerpts, the following behavior can be observed: First, at time 0.3 (line 4), the vehicle model receives a message to update the vehicle state (lines 6-8), causing an

```
1  ...
2  <event>
3     <model>ego_vehicle_vehicle</model>
4     <time>0.3</time>
5     <kind>EX</kind>
6     <port name="update_state" category="I">
7        <message>1.0</message>
8     </port>
9     <port name="update_a_wanted" category="I">
10    </port>
11    <state>
12       <mode>update_state</mode>update_state
13    </state>
14 </event>
15 <event>
16    <model>ego_vehicle_vehicle</model>
17    <time>0.3</time>
18    <kind>IN</kind>
19    <port name="vehicle_state" category="O">
20       <message>[16.003999999999998, 20.04,
             0.3999999999999999]</message>
21    </port>
22    <state>
23       <mode>idle</mode>idle
24    </state>
25 </event>
26 <event>
27    <model>ego_vehicle_vehicle</model>
28    <time>0.3</time>
29    <kind>EX</kind>
30    <port name="update_state" category="I">
31    </port>
32    <port name="update_a_wanted" category="I">
33       <message>-3.0</message>
34    </port>
35    <state>
36       <mode>idle</mode>idle
37    </state>
38 </event>
39 ...
```

Listing 2. Excerpt of the ego vehicle log, with PythonPDEVS.

```
1  ...
2  <event>
3     <model>ego_vehicle_vehicle</model>
4     <time>0.30000000000000004</time>
5     <kind>EX</kind>
6     <port name="update_state" category="I">
7        <message>1.0</message>
8     </port>
9     <port name="update_a_wanted" category="I">
10    </port>
11    <state>
12       <mode>update_state</mode>update_state
13    </state>
14 </event>
15 <event>
16    <model>ego_vehicle_vehicle</model>
17    <time>0.30000000000000004</time>
18    <kind>IN</kind>
19    <port name="vehicle_state" category="O">
20       <message>[16.004, 20.04,
             0.40000000000000013]</message>
21    </port>
22    <state>
23       <mode>idle</mode>idle
24    </state>
25 </event>
26 <event>
27    <model>ego_vehicle_vehicle</model>
28    <time>0.30000000000000004</time>
29    <kind>EX</kind>
30    <port name="update_state" category="I">
31    </port>
32    <port name="update_a_wanted" category="I">
33       <message>-3.0</message>
34    </port>
35    <state>
36       <mode>idle</mode>idle
37    </state>
38 </event>
39 ...
```

Listing 3. Excerpt of the ego vehicle log, using our approach.

external transition (line 5) to “update_state” (line 11-13). From “update_state”, there is an instantaneous transition to the “idle” state, whereby a message will be generated with the new vehicle state (position, velocity, and acceleration). However, at the same time, the vehicle model receives a message from the controller with a new wanted acceleration. Hence, the confluent transition function must be executed. As is common, the confluent transition function has been defined as first executing the internal transition function, followed by the external transition function.[19] This can be observed in the logs, whereby the internal transition (line 18) is executed, generating an output message with the new vehicle state (lines 19-21), and transitioning the model to the “idle” state (line 23). Then, an external transition (line 29) is executed, wherein the vehicle model receives the new wanted acceleration (lines 32-34), and transitions back to the “idle” state (line 36).

Comparing both excerpts, it can be observed how the behavior is equivalent between both approaches. Meaning, the same events occur at (nearly) the same points in time, in (exactly) the same order, with (nearly) the same input/output messages. However, the numerical differences between both approaches is also apparent. It can be observed how the timestamps do not match exactly between both approaches (lines 4, 17, and 28). As the vehicle state (position, velocity, and acceleration) is dependent on the elapsed time, this also results in small differences in the “vehicle_state” message (lines 19-21).

When comparing the full logs for all atomic DPEVS models, we find that they are all equivalent between the two approaches. Therefore, the behavior of the atomic models matches in our approach when compared to an established PDEVS simulator, i.e., PythonPDEVS.

While these results confirm that FMI 3.0 can indeed preserve PDEVS semantics for the presented case study, some practical limitations remain. These are discussed in the following section.

Limitations

Although the presented results demonstrate that our approach can preserve PDEVS semantics in FMI 3.0 co-simulations, some limitations remain that may affect applicability in broader settings. In this section, we discuss these and outline potential strategies to address them where possible.

Coupling Restrictions

Many-to-Many Coupling. The current version of the FMI standard only allows a one-to-many structure when connecting inputs and outputs of FMUs. However, PDEVS is more flexible as it allows for many-to-many couplings, where multiple component models can influence each other. Therefore, the use of FMI for co-simulating PDEVS FMUs restricts the allowed couplings. While this restriction does not break correctness for simple examples, it does limit the applicability of the presented approach for more complex co-simulations.

Possible workarounds include the introduction of additional "aggregator FMUs", which could collect events on multiple inputs, combine them, and then forward them to one or more other FMUs using a single output. Alternatively, further semantic adaptation could be performed to achieve a similar effect. More specifically, a PDEVS FMU could be again wrapped into an FMU with multiple inputs, whereby the parent wrapper could perform the aggregation function for the enclosed PDEVS FMU. Similar approaches are described by Gomes et al.[23] to extend FMUs with functionality such as input interpolation.

However, both approaches have similar drawbacks. They add modeling overhead and their concrete implementation (e.g., number of supported inputs, aggregation logic) would depend heavily on the specific co-simulation setup in which they are used, making them application-specific, limiting re-use, and increasing maintenance effort when the co-simulation setup changes.

Translation Functions. Coupled PDEVS models can include a set of translation functions ($\{Z_{i,j}\}$) to translate an output event of one component to a corresponding input event for another component if necessary. This increases the flexibility of the PDEVS formalism as it allows the coupling of component models with otherwise incompatible event definitions. However, in the presented approach, we do not explicitly consider translation functions, essentially assuming them to be identity functions. While sufficient for the presented example, it does limit the flexibility of the presented approach and its applicability to more complex scenarios that require event translation.

As these translation functions are defined at the level of the coupled models, these also need to be considered at the level of the co-simulation. The FMI standard itself has no specific provisions for adapting incompatible signals. As with the many-to-many coupling, such functionality could be introduced through additional semantic adaptation. Gomes et al.[23] describe such an approach to introduce unit conversions between FMUs to solve signal data mismatches. While this would restore flexibility, it again comes at the cost of additional modeling overhead and limited re-use.

Hierarchical Modeling

A core strength of the PDEVS formalism is its support for hierarchical modeling, where coupled models can contain both atomic models and other coupled models. This allows large systems to be organized into nested coupled models, improving modularity and maintainability. However, FMI co-simulation is inherently flat as FMUs are coordinated by a central orchestration algorithm without an explicit notion of hierarchy. As such, once PDEVS models are wrapped into FMUs, this hierarchical modeling capability is lost.

This loss of hierarchical modeling capability when using FMI has practical drawbacks to modelers. Particularly, it makes maintaining, modifying, and debugging models more difficult as the structure of the modeled system is not easily visible in the co-simulation. However existing PDEVS debugging approaches[33] may provide useful support in this regard.

A potential strategy to reintroduce hierarchical modeling is to adopt a hierarchical co-simulation technique, as described by Gomes et al.[23] and supported in the FMPy[30] tool. In their approach, a set of coupled "internal" FMUs is wrapped into a higher-level "external" FMU, such that the entire assembly behaves as a standard FMU from the perspective of the orchestration algorithm. While this reintroduces hierarchical modeling in an FMI-based co-simulation, it again requires additional modeling effort and increases implementation complexity.

Numerical Precision and Time-Keeping

Another limitation concerns numerical precision and how it affects time-keeping and simulation results. In FMI co-simulations, time and step sizes are communicated using floating-point numbers. As in most simulation setups, this can introduce small rounding errors, which can accumulate over time if not managed carefully. In our results, we also observed such discrepancies when comparing our FMI-based approach to PythonPDEVS. While these discrepancies were small and did not meaningfully affect the simulated behavior of the system, they do complicate strict reproducibility across different simulation tools.

It is important to note that these difficulties are not unique to our approach, or FMI in general, but affect most simulators that rely on floating-point arithmetic. In fact, FMI includes different mechanisms to detect and mitigate the potential of time-drift or mismatched time between FMUs. First, the *doStep* call communicates both the current simulated time and the requested step size to an FMU, allowing it to detect mismatches between FMU and orchestrator, and the FMU can opt for calculating the next time as $current\_time + step\_size$ instead of just accumulating $step\_size$, which would indeed introduce time-drift. Second, the synchronous clocks introduced in FMI 3.0[34] provide an additional mechanism for time-synchronization. While an FMU can communicate to the importer when a particular time-based input clock *should* tick, it is up to the importer to activate the clocks at the correct moment in time. Here, the importer is considered as the single source of truth for simulation time. When a clock ticks, the FMU is expected to synchronize to the corresponding time. Additionally, fixed-point arithmetic may be used to further manage time-keeping, e.g., by defining a fixed time-base, in situations where strict reproducibility is critical.

Together, these mechanisms help to manage time-drift and resulting numerical discrepancies. However, it should be noted that perfect reproducibility across different simulators is difficult to guarantee in any floating-point-based environment.

Continuous Dynamics and Hybrid Simulations

In the current paper, we focused exclusively on discrete-event models. While we demonstrated that FMI 3.0 can preserve PDEVS semantics, we have not yet considered hybrid settings that combine PDEVS FMUs with continuous-time FMUs. This is a limitation as many practical applications, such as the simulation of cyber-physical systems, require the co-simulation of both continuous dynamics and discrete-event-driven components.

In such a hybrid setting, the events produced by the PDEVS FMUs must be correctly deserialized and interpreted by continuous FMUs, while state changes in continuous FMUs must be translated into events the PDEVs FMUs can understand. While FMI 3.0 provides mechanisms such as synchronous clocks and event mode to support such interoperability, these do not by themselves solve this communication.

One way to provide a bridge between continuous and discrete-event components is to introduce explicit interface models. Jiresal and Wainer[35] formalize cyber-physical system interfaces in DEVS using dedicated sensor and actuator models. In their work, the sensor models translate physical sensor signals into discrete events that their DEVS models can process, while actuator models translate events into physical signals. A similar approach could be taken in FMI-based co-simulations, where interface FMUs would take on the role of translating between continuous signals and PDEVS events. Such constructs could provide a structured way to bridge PDEVS and continuous-time FMUs, though at the cost of additional modeling effort.

Use of PDEVS over DEVS

A possible limitation of our approach is that it relies on PDEVS rather than the original DEVS formalism. This choice was necessary because PDEVS solves fundamental challenges in coupling models, in particular it avoids reliance on a global select function to resolve simultaneous events. This proved essential to correctly couple models using FMI 3.0 without relying on a DEVS-specific orchestration algorithm or other model-specific coordination mechanisms.

In practice, the impact of this choice is limited. Many existing models are already defined in PDEVS, requiring no additional effort. For models defined in classic DEVS, migration to PDEVS should be straightforward, as PDEVS is a conservative extension of DEVS. The actual cost–benefit trade-off will depend on the specific scenario, but in cases where FMI-based co-simulation is needed, the additional effort of migration is generally manageable relative to the advantages offered by this approach.

Together, the identified limitations define the boundaries of the presented contribution. They emphasize both the assumptions under which PDEVS semantics can be preserved in FMI 3.0 and the challenges that remain for broader applicability. The demonstrated preservation of PDEVS semantics confirms the feasibility of the approach, while the identified challenges highlight promising directions for extending it. The next section summarizes our contributions and outlines directions for future research.

Conclusions and Future Work

To summarize, we have presented an approach to couple DEVS models using the FMI standard 3.0 making heavy use of the new synchronous clocks. We have shown how Parallel DEVS can be used to avoid the need for a select function when coupling DEVS models through FMI. We have also shown how the different steps in the PDEVS abstract simulator map to different FMI functions, and how an FMI orchestration algorithm can be implemented to ensure the correct coupling of DEVS FMUs. Lastly, we have presented a common example orchestration algorithm based on the literature that meets these requirements.

It is worth stressing the importance of bridging these two formalisms. As shown by Vangheluwe,[3] many hybrid and discrete formalisms can be transformed into DEVS. By enabling the coupling of DEVS models using FMI 3.0, we open up new possibilities for hybrid system simulation, allowing for broader interoperability and more flexible co-simulation setups across different modeling paradigms.

As limitations, we have chosen to not validate the work with combinations of non-PDEVS FMUs and PDEVS FMUs. Since we obey the FMI standard interface, we do not lose generality in assuming all FMUs are PDEVS FMUs. A minor limitation is in the serialization of events that forces other FMUs to parse them accordingly. It is trivial to adjust this to ensure interoperability with a set of known FMUs, but impossible to ensure its compatibility with *any* FMU.

A more significant limitation, as previously mentioned, is that FMI cannot fully preserve the coupling semantics of PDEVS models in all cases. Specifically, FMI only supports "one-to-many" connections, whereas PDEVS allows "many-to-many" coupling. This restriction impacts the correct propagation of simultaneous events across interconnected FMUs in specific scenarios. Additionally, in adapting PDEVS models into FMUs, we lose PDEVS' ability for hierarchical composition of models, and floating-point time communication can introduce small discrepancies that complicate strict reproducibility. Finally, our contribution has so far only addressed discrete-event models, leaving open the question of how PDEVS FMUs can be co-simulated with continuous-time FMUs. Together, these limitations define the boundaries of the presented approach, but the demonstrated preservation of PDEVS semantics confirms its feasibility and motivates further extension.

In future work, we aim to address these challenges. Extending FMI-based co-simulation with support for many-to-many couplings and translation functions will likely require additional semantic adaptations of PDEVS FMUs.[23] We also plan to investigate wrapper-based techniques for reintroducing hierarchical modeling.

Most importantly, we intend to explore the integration of PDEVS FMUs with continuous-time FMUs to enable hybrid system simulation, such as the cyber-physical systems application shown in previous work.[8] This will require addressing the event serialization to allow interoperability with non-DEVS FMUs, potentially through structured interface models such as sensors and actuators that translate between discrete events and continuous signals.[35]

Finally, we plan to evaluate the computational overhead introduced by coupling DEVS models through FMI.

Using benchmarks such as DEVStone,[36] we will assess performance trade-offs and scalability of the approach.

Acknowledgements

GPT-4 and GPT-5 were used to improve the clarity and structure of this paper, particularly in refining explanations and enhancing readability. Additionally, GPT-4 assisted in generating and refining sections of code used for obtaining and analyzing the results. However, all intellectual contributions, analyses, and conclusions are the authors' own, and they take full responsibility for the final content.

Statements and Declarations

Ethical considerations

This article does not contain any studies with human or animal participants.

Consent to participate

Not applicable.

Consent for publication

Not applicable.

Declaration of conflicting interest

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding statement

This research was supported by Flanders Make, the strategic research center for the manufacturing industry, within the Flexible Multi-Domain Design for Mechatronic Systems (FlexMoSys) project. In addition, part of this work has been funded and supported by the DIGIT-Bench project (case no. 640222-497272), funded by the Energy Technology Development and Demonstration Programme (EUDP).

Data availability

All code and models required to reproduce the results of this article are openly available at: https://github.com/Cosys-Lab/2025-SIMULATION-DEVS-FMI3.0.

References

1. Zeigler BP. *Theory of Modelling and Simulation*. New York, Wiley, 1976. ISBN 0-471-98152-4.
2. Chow ACH and Zeigler BP. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Proceedings of Winter Simulation Conference*. IEEE, pp. 716–722.
3. Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *International Symposium on Computer-Aided Control System Design (Cat. No.00TH8537)*. Anchorage, AK, USA: IEEE. ISBN 0-7803-6566-6, pp. 129–134. DOI:10.1109/CACSD.2000.900199.
4. Gomes C, Thule C, Broman D et al. Co-simulation: A Survey. *ACM Computing Surveys* 2018; 51(3): 49:1–49:33. DOI: 10.1145/3179993.
5. Gomes C, Najafi M, Sommer T et al. The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations. In *Proceedings of the 14th International Modelica Conference*. online: Linköping University Electronic Press, Linköpings Universitet, pp. 27–36. DOI:10.3384/ecp2118127.
6. Hansen ST, Thule C, Gomes C et al. Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps. *Int J Softw Tools Technol Transfer* 2022; 24(6): 999–1024. DOI:10.1007/s10009-022-00686-8.
7. Hansen ST, Gomes C and Kazemi Z. Synthesizing Orchestration Algorithms for FMI 3.0. In *2023 Annual Modeling and Simulation Conference*. Ontario, Canada, pp. 184–195.
8. Vanommeslaeghe Y, Acker BV, Denil J et al. Integrating devs and fmi 3.0 for the simulated deployment of embedded applications. In *2024 Annual Modeling and Simulation Conference (ANNSIM)*. Washington, DC, USA: to appear, p. to appear.
9. The MathWorks, Inc. Adaptive Cruise Control System Using Model Predictive Control. URL https://www.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html.
10. Van Tendeloo Y and Vangheluwe H. PythonPDEVS: a distributed parallel DEVS simulator. In *SpringSim (TMS-DEVS)*. pp. 91–98.
11. Blochwitz T, Otter M, Arnold M et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*. Dresden, Germany: Linköping University Electronic Press; Linköpings universitet, pp. 105–114. DOI:10.3384/ecp11063105.
12. Junghanns A, Blochwitz T, Bertsch C et al. The Functional Mock-up Interface 3.0 - New Features Enabling New Applications. In *Proceedings of the 14th International Modelica Conference*. online: Linköping University Electronic Press, Linköpings Universitet, pp. 17–26. DOI:10.3384/ecp2118117.
13. Hansen ST, Gomes C, Najafi M et al. The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations. *Electronics* 2022; 11(21): 3635. DOI:10.3390/electronics11213635.
14. Bastian J, Clauß C, Wolf S et al. Master for Co-Simulation Using FMI. In *8th International Modelica Conference*. Dresden, Germany: Linköping University Electronic Press, Linköpings universitet, pp. 115–120. DOI:10.3384/ecp11063115.
15. Broman D, Brooks C, Greenberg L et al. Determinate composition of FMUs for co-simulation. In *Eleventh ACM International Conference on Embedded Software*. Montreal, Quebec, Canada: IEEE Press Piscataway, NJ, USA. ISBN 978-1-4799-1443-2, p. Article No. 2.
16. Ravi S, Beermann L, Kotte O et al. Timing-Aware Software-in-the-Loop Simulation of Automotive Applications with FMI 3.0. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. pp. 62–72. DOI:10.1109/MODELS58315.2023.00022.
17. Vangheluwe H. The discrete event system specification (DEVS) formalism. *Course Notes, Course: Modeling and Simulation (COMP522A), McGill University, Montreal Canada* 2001; 13.

18. Van Tendeloo Y and Vangheluwe H. Classic DEVS modelling and simulation. In *2017 Winter Simulation Conference (WSC)*. IEEE, pp. 644–658.
19. Van Tendeloo Y and Vangheluwe H. Introduction to parallel DEVS modelling and simulation. In *SpringSim (Mod4Sim)*. pp. 10–1.
20. Chow AC, Zeigler BP and Kim DH. Abstract simulator for the parallel DEVS formalism. In *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. IEEE, pp. 157–163.
21. Vangheluwe H, De Lara J and Mosterman PJ. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AI, Simulation and Planning in High Autonomy Systems Conference*. Lisbon, Portugal: Society for Computer Simulation International, pp. 9–20.
22. Mustafiz S, Gomes C, Barroca B et al. Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Pasadena, California: IEEE, pp. 29:1–29:8. DOI:10.23919/TMS.2016.7918835.
23. Gomes C, Meyers B, Denil J et al. Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators. *SIMULATION* 2018; 95(3): 1–29. DOI:10.1177/0037549718759775.
24. Camus B, Galtier V, Caujolle M et al. Hybrid Co-simulation of FMUs using DEV&DESS in MECSYCO. In *Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium (TMS/DEVS 16)*. Pasadena, CA, United States: Society for Computer Simulation International San Diego, CA, USA, p. No. 8.
25. Quraishi MH, Sarjoughian HS and Gholami S. CO-SIMULATION OF HARDWARE RTL AND SOFTWARE SYSTEM USING FMI. In *2018 Winter Simulation Conference (WSC)*. pp. 572–583. DOI:10.1109/WSC.2018.8632395.
26. Lin X. Co-simulation of Cyber-Physical Systems Using DEVS and Functional Mockup Units. Technical report, Arizona State University, 2021.
27. Joshi R, Nutaro J, Zeigler B et al. Functional Mock-up Interface Based Simulation of Continuous-Time Systems in Cadmium. In *2024 Annual Modeling and Simulation Conference (ANNSIM)*. pp. 1–15. DOI:10.23919/ANNSIM61499.2024.10732622.
28. Vanommeslaeghe Y, Van Acker B, Vanherpen K et al. A co-simulation approach for the evaluation of multi-core embedded platforms in cyber-physical systems. In *Proceedings of the 2020 Summer Simulation Conference*. SummerSim '20, San Diego, CA, USA: Society for Computer Simulation International. ISBN 978-1-7138-1429-0, pp. 1–12.
29. Paris T, Ciarletta L and Chevrier V. A component approach for DEVS. In *Proceedings of the 50th Computer Simulation Conference*. Proceedings of the 50th Computer Simulation Conference, Bordeaux, France: Society for Computer Simulation International, p. 30.
30. CATIA Systems. FMPy. URL https://github.com/CATIA-Systems/FMPy.
31. Legaard CM, Tola D, Schranz T et al. A universal mechanism for implementing functional mock-up units. In *11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. SIMULTECH 2021, Virtual Event, pp. 121–129.
32. Chen B and Vangheluwe H. Symbolic flattening of DEVS models. In *SummerSim*. Citeseer, pp. 209–218.
33. Van Mierlo S, Van Tendeloo Y and Vangheluwe H. Debugging parallel DEVS. *Simulation* 2017; 93(4): 285–306.
34. The Modelica Association. Functional mock-up interface specification, 2022. URL https://fmi-standard.org/docs/3.0/.
35. Jiresal RS and Wainer GA. Formalizing cyber-physical system interfaces using DEVS. In *ANNSIM*. pp. 159–170.
36. Glinsky E and Wainer G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*. IEEE, pp. 265–272.

Author Biographies

Yon Vanommeslaeghe is a postdoctoral researcher at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is a co-worker at Flanders Make. His research interests include modeling and simulation for optimal (embedded) deployment in the context of cyber-physical systems.

Cláudio Gomes is an Assistant Professor at Aarhus University in the Department of Electrical and Computer Engineering. His research interests include co-simulation, digital twin engineering, and machine learning for digital twins, with a focus on ensuring reliability and accuracy in complex cyber-physical systems.

Bert Van Acker is a postdoctoral researcher at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is a co-worker at Flanders Make. His research interests include modeling and simulation for optimal deployment in the robotics and automation domain.

Joachim Denil is an Associate Professor at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is associated with Flanders Make. His research interest include the design, verification and evolution of cyber-physical systems.

Paul De Meulenaere is a Professor at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is associated with Flanders Make. His research interests include co-design and software deployment for cyber-physical systems.

Appendix A: Formal Specification of the ACC Models

For completeness, Definitions 1 to 8 provide the formal definition of all atomic and coupled models described in PDEVS Models of the Running Example. The Python-PDEVS implementation of these models is also available on GitHub: https://github.com/Cosys-Lab/2025-SIMULATION-DEVS-FMI3.0.

Notes:

1. For all models, the confluent transition function δ_{conf} is defined as:

$$\delta_{conf}(s, e, x) = \delta_{ext}(\delta_{int}(s), 0, x)$$

2. For state transition functions, updated parts of the state are highlighted in **bold** for clarity
3. The *Vehicle*, *SpeedController*, and *SupervisorPID* atomic models are event-driven, with external events periodically triggering state updates. For these models, the "update_state" mode is not represented in the external transition functions. As there is an instantaneous internal transition from "update_state" to "idle", any events received while in the "update_state" mode will be handled by the confluent transition function. As the confluent transition function is defined as first executing the internal transition function and then the external transition function (see Note 1), the models will always be in the "idle" mode when executing the external transition function. The inclusion of the "update_state" mode is an consequence of the PDEVS formalism, wherein only internal transitions can generate output events.

Vehicle. Definition 1 formalizes the *Vehicle* atomic model, which implements its longitudinal dynamics. The state is defined as $s = (mode, x, v, a, a_w)$, where *mode* is the discrete mode of the vehicle model, x the position, v the velocity, a the acceleration, and a_w the wanted acceleration.

For transitions involving *update_state*, the updated state variables (x', v', a') are obtained by applying a semi-implicit Euler integration step with elapsed time e:

$$\begin{aligned} a' &= a + e \cdot 2(a_w - a) \\ v' &= v + e \cdot a' \\ x' &= x + e \cdot v' \end{aligned}$$

If both *update_state* and *update_a_wanted* are received simultaneously, the integration step is applied first using the current a_w, after which a_w is updated to a_w^{new}. The updated value takes effect in subsequent steps.

Generator. Definition 2 formalizes the *Generator* atomic model. This model produces *update_state* events at a fixed *interval* and is used to periodically trigger state updates in other models. The generator has no input ports, hence both the input set X and the external transition function δ_{ext} are empty. Its behavior is fully determined by the time advance, internal transition, and output functions.

Sine. Definition 3 formalizes the *Sine* atomic model. This model generates a periodic signal according to $v(t) = A \cdot \sin(\omega \cdot t) + offset$, where A is the amplitude, ω the angular frequency, and $offset$ a constant bias. The signal is produced at a specified *interval*.

The state is defined as (t, v), with $t \in \mathbb{R}$ the current internal time and $v \in \mathbb{R}$ the current signal value. At each internal transition, the state is updated to:

$$(t + interval,\ A \cdot \sin(\omega \cdot (t + interval)) + offset)$$

The output function λ emits the current value v at every internal transition. As the *Sine* model has no input ports, both the input set X and the external transition function δ_{ext} are empty. As with the *Generator* model, its behavior is fully determined by the internal transition and output functions.

SpeedController. Definition 4 formalizes the *SpeedController* atomic model. This implements a standard discrete PID controller to controls the ego vehicle's wanted acceleration based on the difference between a desired speed (*setpoint*) and the actual vehicle speed. The state is defined as $(mode, I, e_{prev}, sp, act, out)$, with $mode$ the discrete mode, I the accumulated integral term, e_{prev} the previous error, sp the current speed setpoint, act the measured actual speed of the vehicle, and out the most recently computed control output.

The model can receive events on three input ports:

- *update_state*, which triggers a controller update
- *update_setpoint*, which provides a new desired speed sp^{new}
- *vehicle_state*, which provides the current actual speed act

When an *update_state* event is received, the new controller output is calculated using the standard discrete PID equations with elapsed time e:

$$\begin{aligned} err &= sp - act \\ I' &= I + err \cdot e \\ d_e &= \frac{err - e_{prev}}{e} \\ out' &= K_p \cdot err + K_i \cdot I' + K_d \cdot d_e \\ e_{prev} &= err \end{aligned}$$

The output function λ generates events with the controller output out.

SupervisorPID. Definition 5 formalizes the *SupervisorPID* atomic model. This model supervises the ego vehicle's speed in order to maintain a safe distance to a lead vehicle. In the adaptive cruise control (ACC) scenario, it is responsible for generating updated speed setpoints v_{wanted} for the *SpeedController* model. The state is defined as $(mode, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted})$, with $mode$ the discrete mode, (x_{ego}, v_{ego}) the ego vehicle's position and velocity, (x_{lead}, v_{lead}) the lead vehicle's position and velocity, and v_{wanted} the current wanted velocity. For clarity, additional PID-related state variables such as the integral term I, the previous error e_{prev}, and the current setpoint sp have been omitted from the formal definition, but their role is analogous to the *SpeedController* model described earlier.

The model can receive events on three input ports:

- *update_state*, which triggers a supervisor update
- *ego_vehicle_state*, providing x_{ego} and v_{ego}
- *lead_vehicle_state*, providing x_{lead} and v_{lead}

When an *update_state* event is received, the supervisor updates v_{wanted} based on the relative position and velocity of the ego and lead vehicle. More specifically, v_{wanted} is adjusted using a PID controller based on the actual relative position (gap) between the vehicles and a safe distance, calculated from their relative velocities.

The output function λ generates events with this wanted velocity v_{wanted}.

EgoVehicle. Definition 6 formalizes the *EgoVehicle* coupled model. It consists of a *Vehicle* atomic model and a *Generator* that periodically issues *update_state* events. The coupled model has an input port *update_a_wanted* for the wanted acceleration and outputs the current vehicle state on the *vehicle_state* port.

LeadVehicle. Definition 7 formalizes the *LeadVehicle* model. It combines *Vehicle*, *Generator*, and *Sine* atomic model. The *LeadVehicle* model works similar to the *EgoVehicle* model, with the *Generator* periodically issuing *update_state* events for the *Vehicle*. In this case however, the wanted acceleration is provided by the *Sine* model, which represents the driver of the lead vehicle. This model also outputs the current vehicle state on its *vehicle_state* port.

AdaptiveCruiseControlSystem. Definition 8 formalizes the full *AdaptiveCruiseControlSystem*. It integrates the *EgoVehicle*, *LeadVehicle*, *SpeedController*, and *Supervisor-PID* models. Here, the supervisor determines speed setpoints for the speed controller based on the relative distance and relative velocity between the ego and lead vehicles. The speed controller then provides a wanted acceleration for the ego vehicle to regulate its speed. Generators provide periodic *update_state* events to both the controller and supervisor to trigger updates. Together, the coupled system captures the closed-loop adaptive cruise control scenario.

$$Vehicle = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle$$

$$S = \{(\boldsymbol{mode}, x, v, a, a_w) \mid \boldsymbol{mode} \in \{idle, update\_state\},\ x, v, a, a_w \in \mathbb{R}\}$$

$$ta = \begin{Bmatrix} (idle, x, v, a, a_w) \rightarrow +\infty, \\ (update\_state, x, v, a, a_w) \rightarrow 0 \end{Bmatrix}$$

$$\delta_{int} = \{(update\_state, x, v, a, a_w) \rightarrow (\boldsymbol{idle}, x, v, a, a_w)\}$$

$$X = X_{update\_state} \times X_{update\_a\_wanted} = \{update\_state\} \times \mathbb{R}$$

$$\delta_{ext} = \begin{Bmatrix} (idle, x, v, a, a_w),\ e,\ (\phi, a_w^{new}) \rightarrow (idle, x, v, a, \boldsymbol{a_w^{new}}), \\ (idle, x, v, a, a_w),\ e,\ (update\_state, \phi) \rightarrow (\boldsymbol{update\_state}, \boldsymbol{x'}, \boldsymbol{v'}, \boldsymbol{a'}, a_w), \\ (idle, x, v, a, a_w),\ e,\ (update\_state, a_w^{new}) \rightarrow (\boldsymbol{update\_state}, \boldsymbol{x'}, \boldsymbol{v'}, \boldsymbol{a'}, \boldsymbol{a_w^{new}}) \end{Bmatrix}$$

$$Y = Y_{vehicle\_state} = \mathbb{R}^3$$

$$\lambda = \begin{Bmatrix} (update\_state, x, v, a, a_w) \rightarrow (x, v, a), \\ (idle, x, v, a, a_w) \rightarrow \phi \end{Bmatrix}$$

Definition 1. Atomic PDEVS specification of the *Vehicle* model.

$$Generator = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle$$

$$S = \{idle\}$$

$$ta = \{idle \rightarrow interval\}$$

$$\delta_{int} = \{idle \rightarrow idle\}$$

$$X = \phi$$

$$\delta_{ext} = \phi$$

$$Y = Y_{update\_state} = \{update\_state\}$$

$$\lambda = \{idle \rightarrow update\_state\}$$

Definition 2. Atomic PDEVS specification of the *Generator* model.

$$Sine = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle$$

$$S = \{(t, v) \mid t, v \in \mathbb{R}\}$$

$$ta = \{(t, v) \rightarrow interval\}$$

$$\delta_{int} = \{(t, v) \rightarrow (\boldsymbol{t + interval},\ \boldsymbol{A \cdot \sin(\omega \cdot (t + interval)) + offset})\}$$

$$X = \phi$$

$$\delta_{ext} = \phi$$

$$Y = Y_{value} = \mathbb{R}$$

$$\lambda = \{(t, v) \rightarrow v\}$$

Definition 3. Atomic PDEVS specification of the *Sine* model.

$$SpeedController = \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle$$

$$S = \{(mode, I, e_{prev}, sp, act, out) \mid mode \in \{idle, update\_state\},\ I, e_{prev}, sp, act, out \in \mathbb{R}\}$$

$$ta = \left\{ \begin{array}{c} (idle, I, e_{prev}, sp, act, out) \rightarrow +\infty, \\ (update\_state, I, e_{prev}, sp, act, out) \rightarrow 0 \end{array} \right\}$$

$$\delta_{int} = \{(update\_state, I, e_{prev}, sp, act, out) \rightarrow (\boldsymbol{idle}, I, e_{prev}, sp, act, out)\}$$

$$X = X_{update\_state} \times X_{update\_setpoint} \times X_{vehicle\_state} = \{update\_state\} \times \mathbb{R} \times \mathbb{R}^3$$

$$\delta_{ext} = \left\{ \begin{array}{l} (idle, I, e_{prev}, sp, act, out),\ e, (\phi, \phi, (x, v, a)) \\ \qquad\qquad \rightarrow (idle, I, e_{prev}, sp, \boldsymbol{v}, out), \\ (idle, I, e_{prev}, sp, act, out),\ e, (\phi, sp^{new}, \phi) \\ \qquad\qquad \rightarrow (idle, I, e_{prev}, \boldsymbol{sp^{new}}, act, out), \\ (idle, I, e_{prev}, sp, act, out),\ e, (update\_state, \phi, \phi) \\ \qquad\qquad \rightarrow (\boldsymbol{update\_state}, \boldsymbol{I'}, \boldsymbol{e'_{prev}}, sp, act, \boldsymbol{out'}), \\ (idle, I, e_{prev}, sp, act, out),\ e, (update\_state, sp^{new}, \phi) \\ \qquad\qquad \rightarrow (\boldsymbol{update\_state}, \boldsymbol{I'}, \boldsymbol{e'_{prev}}, \boldsymbol{sp^{new}}, act, \boldsymbol{out'}), \\ (idle, I, e_{prev}, sp, act, out),\ e, (update\_state, \phi, (x, v, a)) \\ \qquad\qquad \rightarrow (\boldsymbol{update\_state}, \boldsymbol{I'}, \boldsymbol{e'_{prev}}, sp, \boldsymbol{v}, \boldsymbol{out'}), \\ (idle, I, e_{prev}, sp, act, out),\ e, (update\_state, sp^{new}, (x, v, a)) \\ \qquad\qquad \rightarrow (\boldsymbol{update\_state}, \boldsymbol{I'}, \boldsymbol{e'_{prev}}, \boldsymbol{sp^{new}}, \boldsymbol{v}, \boldsymbol{out'}) \end{array} \right\}$$

$$Y = Y_{output} = \mathbb{R}$$

$$\lambda = \left\{ \begin{array}{c} (update\_state, I, e_{prev}, sp, act, out) \rightarrow out, \\ (idle, I, e_{prev}, sp, act, out) \rightarrow \phi \end{array} \right\}$$

Definition 4. Atomic PDEVS specification of the *SpeedController* model.

$$SupervisorPID = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle$$

$$S = \{(mode, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}) \mid mode \in \{idle, update\_state\},\ x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted} \in \mathbb{R}\}$$

$$ta = \begin{Bmatrix} (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}) \rightarrow +\infty, \\ (update\_state, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}) \rightarrow 0 \end{Bmatrix}$$

$$\delta_{int} = \{(update\_state, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}) \rightarrow (\boldsymbol{idle}, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted})\}$$

$$X = X_{update\_state} \times X_{ego\_vehicle\_state} \times X_{lead\_vehicle\_state} = \{update\_state\} \times \mathbb{R}^3 \times \mathbb{R}^3$$

$$\delta_{ext} = \begin{Bmatrix} (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}),\ e, (\phi, \phi, (x_{lead}^{new}, v_{lead}^{new}, a_{lead}^{new})) \\ \qquad \rightarrow (idle, x_{ego}, v_{ego}, \boldsymbol{x_{lead}^{new}}, \boldsymbol{v_{lead}^{new}}, v_{wanted}), \\ (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}),\ e, (\phi, (x_{ego}^{new}, v_{ego}^{new}, a_{ego}^{new}), \phi) \\ \qquad \rightarrow (idle, \boldsymbol{x_{ego}^{new}}, \boldsymbol{v_{ego}^{new}}, x_{lead}, v_{lead}, v_{wanted}), \\ (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}),\ e, (update\_state, \phi, \phi) \\ \qquad \rightarrow (\boldsymbol{update\_state}, x_{ego}, v_{ego}, x_{lead}, v_{lead}, \boldsymbol{v'_{wanted}}), \\ (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}),\ e, (update\_state, \phi, (x_{lead}^{new}, v_{lead}^{new}, a_{lead}^{new})) \\ \qquad \rightarrow (\boldsymbol{update\_state}, x_{ego}, v_{ego}, \boldsymbol{x_{lead}^{new}}, \boldsymbol{v_{lead}^{new}}, \boldsymbol{v'_{wanted}}), \\ (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}),\ e, (update\_state, (x_{ego}^{new}, v_{ego}^{new}, a_{ego}^{new}), \phi) \\ \qquad \rightarrow (\boldsymbol{update\_state}, \boldsymbol{x_{ego}^{new}}, \boldsymbol{v_{ego}^{new}}, x_{lead}, v_{lead}, \boldsymbol{v'_{wanted}}), \\ (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}),\ e, (update\_state, (x_{ego}^{new}, v_{ego}^{new}, a_{ego}^{new}), (x_{lead}^{new}, v_{lead}^{new}, a_{lead}^{new})) \\ \qquad \rightarrow (\boldsymbol{update\_state}, \boldsymbol{x_{ego}^{new}}, \boldsymbol{v_{ego}^{new}}, \boldsymbol{x_{lead}^{new}}, \boldsymbol{v_{lead}^{new}}, \boldsymbol{v'_{wanted}}) \end{Bmatrix}$$

$$Y = Y_{output} = \mathbb{R}$$

$$\lambda = \begin{Bmatrix} (update\_state, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}) \rightarrow v_{wanted}, \\ (idle, x_{ego}, v_{ego}, x_{lead}, v_{lead}, v_{wanted}) \rightarrow \phi \end{Bmatrix}$$

Definition 5. Atomic PDEVS specification of the *SupervisorPID* model.

$$EgoVehicle = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

$$X_{self} = X_{update\_a\_wanted} = \mathbb{R}$$

$$Y_{self} = Y_{vehicle\_state} = \mathbb{R}^3$$

$$D = \{vehicle, generator\}$$

$$\{M_i\} = \begin{Bmatrix} M_{vehicle} = Vehicle, \\ M_{generator} = Generator \end{Bmatrix}$$

$$\{I_i\} = \begin{Bmatrix} vehicle \rightarrow \{self\}, \\ self \rightarrow \{vehicle\}, \\ generator \rightarrow \{vehicle\} \end{Bmatrix}$$

$$\{Z_{i,j}\} = \begin{Bmatrix} Z_{vehicle,self} = \{vehicle\_state \rightarrow vehicle\_state\}, \\ Z_{generator,vehicle} = \{update\_state \rightarrow update\_state\}, \\ Z_{self,vehicle} = \{update\_a\_wanted \rightarrow update\_a\_wanted\} \end{Bmatrix}$$

Definition 6. Coupled PDEVS specification of the *EgoVehicle* model.

$$LeadVehicle = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

$$X_{self} = \phi$$

$$Y_{self} = Y_{vehicle\_state} = \mathbb{R}^3$$

$$D = \{vehicle, generator, sine\}$$

$$\{M_i\} = \left\{ \begin{array}{c} M_{vehicle} = Vehicle, \\ M_{generator} = Generator, \\ M_{sine} = Sine \end{array} \right\}$$

$$\{I_i\} = \left\{ \begin{array}{c} vehicle \rightarrow \{self\}, \\ generator \rightarrow \{vehicle\}, \\ sine \rightarrow \{vehicle\} \end{array} \right\}$$

$$\{Z_{i,j}\} = \left\{ \begin{array}{c} Z_{vehicle,self} = \{vehicle\_state \rightarrow vehicle\_state\}, \\ Z_{generator,vehicle} = \{update\_state \rightarrow update\_state\}, \\ Z_{sine,vehicle} = \{value \rightarrow update\_a\_wanted\} \end{array} \right\}$$

Definition 7. Coupled PDEVS specification of the *LeadVehicle* model.

$$AdaptiveCruiseControlSystem = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

$$X_{self} = \phi$$

$$Y_{self} = \phi$$

$$D = \{ego, lead, controller, controller\_gen, supervisor, supervisor\_gen\}$$

$$\{M_i\} = \left\{ \begin{array}{c} M_{ego} = EgoVehicle, \\ M_{lead} = LeadVehicle, \\ M_{controller} = SpeedController, \\ M_{controller\_gen} = Generator, \\ M_{supervisor} = SupervisorPID, \\ M_{supervisor\_gen} = Generator \end{array} \right\}$$

$$\{I_i\} = \left\{ \begin{array}{c} controller \rightarrow \{ego\}, \\ controller\_gen \rightarrow \{controller\}, \\ ego \rightarrow \{controller, supervisor\}, \\ lead \rightarrow \{supervisor\}, \\ supervisor \rightarrow \{controller\}, \\ supervisor\_gen \rightarrow \{supervisor\} \end{array} \right\}$$

$$\{Z_{i,j}\} = \left\{ \begin{array}{c} Z_{controller\_gen,controller} = \{update\_state \rightarrow update\_state\}, \\ Z_{ego,controller} = \{vehicle\_state \rightarrow vehicle\_state\}, \\ Z_{controller,ego} = \{output \rightarrow update\_a\_wanted\}, \\ Z_{supervisor\_gen,supervisor} = \{update\_state \rightarrow update\_state\}, \\ Z_{ego,supervisor} = \{vehicle\_state \rightarrow ego\_vehicle\_state\}, \\ Z_{lead,supervisor} = \{vehicle\_state \rightarrow lead\_vehicle\_state\}, \\ Z_{supervisor,controller} = \{output \rightarrow update\_setpoint\} \end{array} \right\}$$

Definition 8. Coupled PDEVS specification of the *AdaptiveCruiseControlSystem* model.