Co-simulation of Cyber-Physical Systems

Using DEVS and Functional Mockup Units

by

Xuanli Lin

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved December 2020 by the
Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Giulia Pedrielli
Guoliang Xue

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Cyber-Physical Systems (CPS) are becoming increasingly prevalent around the world. Co-simulation of cyber and physical components has shown to be an effective way towards the development of time-sensitive and reliable CPS. Correctly combining continuous models with discrete models for co-simulation can often be challenging. In this thesis, the Functional Markup Interface (FMI) is used to develop an adapter called DEVS-FMI for the DEVS-Suite simulator. The adapter, implemented using JavaFMI 2.0, allows any Functional Mock-Up Unit (FMU) to be co-simulated with a Discrete Event System Specification (DEVS) model. This approach enables taking advantage of the parallel DEVS formalism to model cyber systems and using Modelica to model physical systems. An FMU serves as a slave simulator while the DEVS-Suite serves as a master simulator. The Four-Variable model is used as a guide to define the requirements for the inputs and outputs of actuator and sensor devices used in cyber and physical systems. The input and output data as non-functional abstractions of the sensor and actuator devices. Select cyber and physical parts of an electric scooter are chosen, modeled, simulated, and evaluated using the integrated OpenModelica and the DEVS-Suite simulators. Closely related research is briefly examined and expanding this work with support for implicit state-changes for continuous models and distributed co-simulation is noted.

ACKNOWLEDGEMENTS

I would like to thank Dr. Hessam Sarjoughian for his mentorship throughout this research project. I also would like to thank my family for their emotional and financial support along the way. Finally, I would like to give thanks to my graduate committee members, Dr. Giulia Pedrielli and Dr. Guoliang Xue for serving on the thesis committee.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

APPENDIX Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

1. Introduction

CPS are transforming the world and revolutionizing people's lives. CPS can be small things such as Internet of Things (IoT) devices, or big things as Smart Cities and Smart Grids. The interleaving of cyber components and physical components can introduce significant challenges in specifying, designing, prototyping, testing, and validating the system. These challenges necessitate the co-simulation of physical and cyber parts of the system [1].

The FMI [2] is a standard that allows exchange and co-simulation of simulation models produced by different tools that supports the FMI standard. Models produced according to the FMI standard are called Functional Mock-Up Units (FMU). In essence, a FMU is a self-contained package that includes standardized APIs, their underlying implementations, and some metadata of the APIs. FMUs are typically used for modeling continuous components where underlying relationships can be expressed in mathematical equations. Each FMU is a fundamental unit that can be combined into a complex system through partitioning and hierarchy. The FMI standard is widely adopted in various industries and disciplines, and OpenModelica [3], an open-source Modelica development environment, provides full support on FMU export and import.

DEVS formalism [4] lays the foundation of modeling and analyzing event-driven dynamical systems. The fundamental building block in a DEVS model is an atomic model, where a system's time-based behavior can be captured. The atomic model is defined in terms of input and output events, possible states, time, dynamics due to

internal and external events, and output event generation. A coupled model is made up of hierarchically atomic and coupled models with information on how the models are coupled together. DEVS-Suite [5] is a DEVS simulator environment that supports development, execution, experimentation, and visualization of parallel DEVS coupled models.

The goal of DEVS-FMI is to extend upon the existing DEVS-Suite framework to provide interoperability with FMUs, hence achieving co-simulation for CPS. Widl and Müller proposed a generic tool coupling scheme [6] that can be used to enable FMI co-simulation on tools that do not natively support FMI standard (in this case, DEVS-FMI). In essence, an FMI adapter that expands upon DEVS-Suite and supports FMI standard is created to be a middle compatibility layer. To achieve this end, JavaFMI library [7] is used to streamline interactions with FMUs in Java language. Furthermore, the adapter makes use of the Four-Variable model [8] that guides defining inputs and outputs between cyber and physical parts. The rest of this thesis focuses on the inner workings of this adapter and demonstrate its capabilities using a small portion of an electric scooter example.

2. Contributions

Prior research has explored several different approaches to co-simulate cyber and physical components in CPS, and many were proved to be useful in constructing a methodological wrapper for CPS co-simulation. In particular, [9] proposed an approach that extends the DEVS-Suite simulator to support co-simulation using generated FMUs. However, this approach was developed for FMI 1.0 standard and used for co-simulating a circular buffer for Network-on-Chip and an untimed script mimicking a firmware for choosing routing paths.

This thesis presents an improved solution based on the architecture shown in [9], that would lend to prototype FMI 2.0 wrapper interface for co-simulating Parallel DEVS models with FMU. A round-robin synchronization protocol is used to extend the DEVS abstract simulator protocol for co-simulation of the cyber and physical models. The co-simulation approach is implemented as an extension to the DEVS-Suite simulator. Using the Four-Variable model, the inputs and outputs of a simplified electric scooter is identified. The model of the electric scooter is specified and developed using OpenModelica and DEVS-Suite simulators and evaluated to demonstrate the feasibility and effectiveness of the solution proposed in this thesis.

3. Background

There are a number of specifications and frameworks that are used in the development of the DEVS-FMI, including DEVS, FMI, CPS, and Four-Variable model paradigm. The following sections provide a brief overview of these concepts.

3.1. Discrete Event Systems

DEVS formalism provides a modular approach to modeling and analyzing systems. It is used for modeling discrete-event, discrete-time, and continuous time models [4] [10]. DEVS is inherently a timed event system, where inputs (events) and internal timer shape the reactions from the system, ultimately deciding the output. DEVS is also hierarchical – a complete system can be divided into smaller subsystems, which in turn be composed of individual components. Likewise, a top-level DEVS model (coupled model) can be made up of several sub-models (coupled models), which consists of combination of basic models (atomic models). The combination can be arbitrary to suit the system in question. Parallel DEVS is an extension to DEVS that allows for more flexibility when two events (e.g., an external and an internal event) are scheduled to be taking place at the same time [11]. In DEVS-Suite, parallel DEVS is used, and its basic constructs are explained below.

3.1.1. Atomic Models

Formally, an atomic model in parallel DEVS is defined as an 8-tuple [11]

$$M = < X, Y, S, \delta_{ext}, \delta_{int}, \delta_{\text{con}}, \lambda, ta >$$

where,

- X is the set of input events (port-value pairs),
- Y is the set of output events (port-value pairs),
- S is the set of sequential states,
- $\delta_{ext}: Q \times X^b \rightarrow S$ is the external state transition function, where Q corresponds to the current state and time elapsed since last transition,
- $\delta_{int}: S \rightarrow S$ is the internal state transition function,
- $\delta_{con}: Q \times X^b \rightarrow S$ is the confluent function,
- $\lambda: S \rightarrow Y^b$ is the output function,
- $ta: S \rightarrow R_0^+ \cup \infty$ is the time advance function (nonnegative real values and positive infinity are accepted).

A DEVS model is in a state $s \in S$ in any given time. It will stay in the state till $ta(s)$ (that is, the lifetime of the state s) expires, unless perturbed by one or more external events. Once the state expires, it will produce an output prescribed by $\lambda(s)$ and update the state as specified in $\delta_{int}(s)$. External events can trigger external transitions defined by $\delta_{ext}(s, e, X^b)$, where s is the current state, e is time elapsed since last transition, and X^b is a bag of input events. In parallel DEVS, instead of single input and output events, the notion of bags is introduced, which is essentially a set of input or output events. This way, multiple events can be processed simultaneously. When $ta(s) = 0$, state transition is instantaneous and when $ta(s) = \infty$, the model will stay in the state s indefinitely, unless perturbed by one or more external events. Another alteration of

parallel DEVS is the confluent function, which determines the new state of the model when internal and external functions occur at the same time.

Input port(s) are the sole channels of interaction for any given model from an outside perspective, likewise output port(s) are sole channels of response from the model. This ensures a uniform control and access to the model, while keeping the model well encapsulated, thus improving the reusability of a model.

3.1.2. Coupled Models

Multiple atomic models and/or coupled sub-models can be coupled together to form coupled models. It is defined as a 7-tuple [11]

$$CM = < X, Y, D, M_d, EIC, EOC, IC >$$

where,

- X is the set of input events (port-value pairs),
- Y is the set of output events (port-value pairs),
- D is the set of components names for each $d \in D$,
- $M_d : d \in D$ is a constituent model (atomic or coupled),
- EIC is the set of external input couplings, where an input port(s) of the coupled model is linked with input port(s) of its constituent model(s),
- EOC is the set of external output couplings, where an output port(s) of the coupled model is linked with output port(s) of its constituent model(s),
- IC is the set of internal couplings.

3.1.3. Timing

It is worth noting that the DEVS formalism is not tied to real time (i.e., wall clock) – even the concept of timing is prevalent in DEVS constructs, it is used to capture the ordering of the events with infinite accuracy (i.e., logical clock). The speed of the simulation can vary depending on many factors, such as hardware computing platform as well as scale and complexity of the model [12]. DEVS simulator would not account for differences in simulation speed or when input arrives.

3.1.4. DEVS-Suite

The complexity associated with DEVS modeling and experimentation lead to the creation of the DEVS-Suite simulator [5] [13], a DEVS modeling, simulation, and tracking environment. Through DEVS-Suite, one can understand a rather complex DEVS model in an intuitive graphical presentation. It enables observing the model's input/output trajectories, state changes over time, and interacting with the model at runtime.

Currently, DEVS-Suite natively supports parallel DEVS and cellular automata models. Despite earlier works on enabling co-simulation of FMUs based on FMI 1.0 standard [9], the solution is not compatible with FMI 2.0. The earlier development includes a design for co-simulation of DEVS with FMU compliant with FMI 1.0. This approach was demonstrated for Register Transfer Level (RTL) DEVS and a Matlab FMU. This thesis provides an improved design and uses FMI 2.0 with an OpenModelica FMU.

3.2. Functional Mock-up Interface & Modelica

A common problem that troubled many model designers and developers is that tools and languages from different vendors are not necessarily compatible with each other. Creating compatibility layers between every single one of them is unrealistic – there can be numerous combinations of these tools, and these layers must be kept up to date at all time. To solve this challenge for simulation tools, FMI [2] was created to provide a standard for model exchange and co-simulation between different vendors. Figure 1 shows a use-case of FMI in automotive industry.

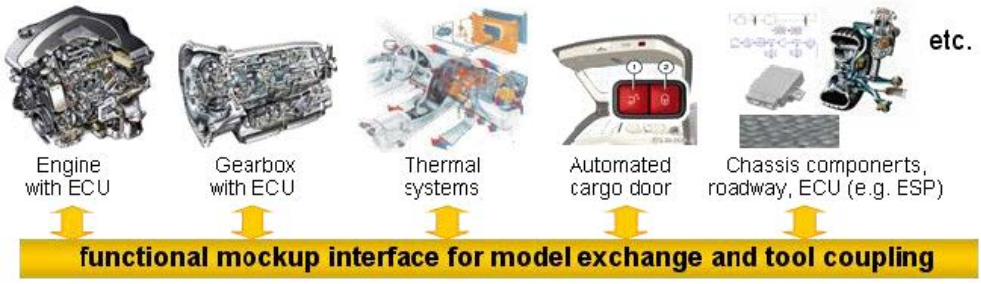

Figure 1. Possible uses of FMI in automotive industry, showing its versatile uses across many disparate needs and requirements *[14]*

3.2.1. Basic Ideas

There are two types of FMI standard as of now: FMI for model exchange, where the model is mainly intended to be exchanged (exported/imported) between different tools; FMI for Co-Simulation, where models from different tools are used to be simulated together under the master and slave simulators concept. For model exchange, the exported module contains only the model itself, and the hosting tool would provide the solver needed to simulate the model. For co-simulation, the module would contain both

the model and the solver, so a co-simulating environment can readily interact with the model without needing to have knowledge of the underlying model.

3.2.2. Functional Mockup Unit

A FMU is simply a component that implements the FMI standard interface. It is a zipped file that contains metadata, implementation of the interface, and additional data and functionality. Figure 2 shows folder structure of a FMU generated by Modelica, note that platform-specific (Windows) binaries were generated, along with the source code and XML metadata.

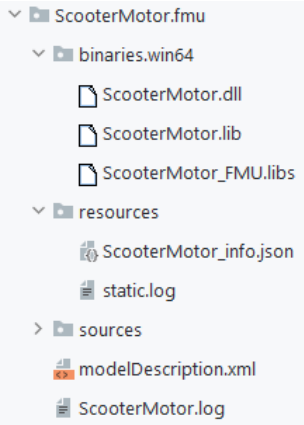

Figure 2. Folder structure inside a FMU generated by Modelica

The metadata defined in XML format includes important information regarding the interface definition, including variable types, variables and their attributes, dependency information, and many more. The interface is implemented in C, though users do not necessarily need to know its details, as all functionalities can be accessed with the interface.

The interface contains the following methods

- Instantiation, initialization, termination, and destruction of FMU.
    - fmiComponent fmiInstantiate (fmiString instanceName, ...)
- Getter and setter functions for each supported type (real, integer, boolean, and string).
    - fmiStatus fmiSetReal (fmiComponent c, const fmiValueReference vr [], size_t nvr, const fmiReal value[])

Some parameters, such as fmiValueReference, can be obtained from the XML descriptor.

3.2.3. Tool Integration

Edmund Widl and Wolfgang Müller proposed a generic architecture for integrating simulation tools that do not provide co-simulation and FMI support natively [6]. This is approach is adopted in the FMI++ library. This architecture consists of two parts: the frontend and the backend components.

- Frontend interfaces with FMI master algorithm. It handles initialization and mapping of variables of the FMUs it interfaces with.
- Backend interfaces with slave (external) applications via an adapter.
    - The adapter is a part of the model loaded in external simulator. It facilitates data exchange between external tool and frontend.
- Frontend and backend are connected via a dedicated data manager.
    - The data manager contains separate interfaces for master and slave.

- Contains functions to enable internal communication between two ends and handles synchronization.

3.2.4. Modelica and OpenModelica Simulator

Modelica is an object-oriented, multi-domain modeling language that is designed for modeling complex systems. A Modelica class contains mainly equations that are evaluated throughout the simulation. It may also include algorithmic components, instances of other classes, parameters, initial conditions, among others. It provides support for four built-in types: real, integer, boolean, and string. Other user-defined types may be derived based on the built-ins, specifying its name, unit, range of value, and more. One can also describe physical connections between two physical ports.

```
model Capacitor "Ideal linear electrical capacitor"
  extends Interfaces.OnePort(v(start=0));
  parameter SI.Capacitance C(start=1) "Capacitance";

equation
  i = C*der(v);
  annotation (...);
end Capacitor;
```

Figure 3. A linear capacitor model in Modelica showing some of the language features

Figure 3 above shows a simple linear capacitor model in Modelica, included as part of the standard library. Here, class Capacitor (designated as a model, a special type of class) extends OnePort interface, and introduces a custom typed parameter Capacitance. In

addition, there is a new equation $i = C * der(v)$, which simply indicates that current value is obtained by multiplying the capacitance and derivative of the voltage. Note that the definition for voltage and current, as well as some basic formulae, is in OnePort interface.

It is apparent that Modelica models is expressive and hierarchical. However, it is not meant to be a general-purpose programming language such as Java or Python. As mentioned above, models are generally defined as ordinary differential equations (ODE), partial differential equations (PDE), and differential algebraic equations (DAE) that are evaluated continuously in miniscule time steps. As shown in Figure 4, there is no "compilation" in the general sense, but the model is parsed, flattened, and undergone several optimizations and decompositions, before finally translated into C code and packaged into a FMU [15]. Algorithmic components, such as if-else statements and loops, can be used. However, they are more often used to generate equations with varying parameters than to control the execution logic.

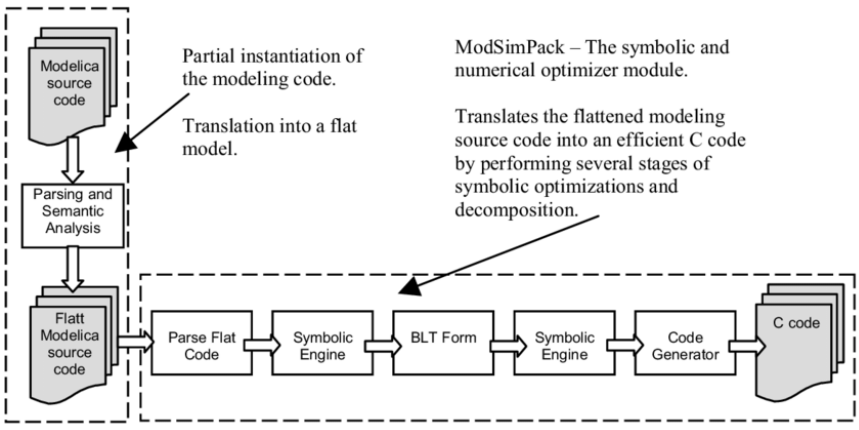

Figure 4. Code generation process in Modelica language *[15]*

OpenModelica [3] is a visual environment for modeling, simulating, and analyzing Modelica models. It contains a Modelica compiler, a connection editor (graphical development environment for models), and various interfaces targeting other programming languages. It also includes the Modelica library, which contains thousands of readymade models for various engineering fields, such as electrical, magnetic, mechanical, fluid, and thermal systems. Complex models can be either viewed graphically to get a top-level view of the system and relationships between their components, or textually understand the model in a fundamental way. OpenModelica also provides built-in plotter that can show the simulation results in graphs.

3.2.5. Timing

Like DEVS, timing is also crucial to Modelica models. Simulation is based on continual evaluation of the equations in the model, however the Modelica models are not necessarily real-time. Simulation is tied to logical clock that are defined in the simulator. Even though the model is defined in terms of continual time, it is extremely expensive to have real continuous simulation (i.e., update equations with infinitesimal time interval), a compromise is to set the time interval between updates (step size) a reasonable number so that the result would still be acceptably accurate, while still computationally feasible.

3.3. Cyber-Physical Systems

A CPS can be thought of a collection of interrelated computing devices and their environment interacting via sensors and actuators [16]. CPS make use of their presence in both cyber and physical worlds to help achieve otherwise difficult tasks by using either

cyber or physical means alone. The world today makes extensive use of CPS – in households, hospitals, roads, electric grids, and much more. The rise of CPS and its extensive use in many critical areas benefits from robust and streamlined modeling and simulation approaches.

3.3.1. Challenges with CPS Design and Analysis

A major challenge arises from the nature of CPS. Since cyber and physical parts of CPS are so closely tied to each other, they must be acting coherently, correctly, and consistently with each other [17]. Since the physical parts is usually expressed in some continuous equations and cyber part is described in discrete models, it is difficult to combine them to form realistic simulations for CPS. In addition, the input can arrive spontaneously, and the underlying components can exhibit nondeterministic behaviors. Therefore, traditional mathematical and analytical models can be too expensive to be used [18].

As CPS are generally complex, designing and testing becomes increasingly difficult for them. For many large-scale CPS deployments, including smart grid and smart city, testing on smaller scales can be still difficult yet provides limited insight. Many devices, such as electrocardiograph (ECG) machines and electrical grid controllers, have very stringent timing requirements and complex dynamics, and it can be laborious to ensure that the product adheres the specification during the design phase.

3.3.2. Benefits of Co-simulation

In recent years, co-simulation has yielded considerable success in tackling challenges associated with CPS design and analysis [1] [19]. Simply put, co-simulation puts models from cyber and physical worlds side by side and simulate them together as one. With co-simulation, it is possible to construct large and sophisticated dynamic systems with a modular and hierarchical approach. It also simplifies observing and controlling the interaction between the cyber and the physical parts, and the system's reaction to internal and external stimuli. This facilitates the understanding of the system in different conditions. Finally, simulation can enable model reuse and rapid prototyping, improving the design and development process.

3.3.3. Considerations for Co-simulation

It is crucial to ensure that the master simulator is constructed correctly to account for requirements of co-simulation.

- Time scale. Typically, physical models are continuous and related to real time. Changes can take place in a very short time span, such as in milliseconds. In contrast, cyber models are often discrete, and updates can be irregular or comparably slower compared to physical counterparts. Therefore, the simulator needs to coordinate between models with two distinct time resolutions and establish correct causality relationships between them.

- Input/output format. Continuous models expect continual input and would produce continual output over time. Discrete systems are often driven by internal and external stimuli (events) that cause some state transitions and produce outputs [4]. The simulator should be able to provide a reliable messaging protocol between the two systems to ensure reliable exchange of information.
- Cyber and discrete components can have different interfaces and operating environments. Even though some standards such as FMI can alleviate this problem, many tools do not support FMI yet. Therefore, it may be necessary to establish some intermediate components to ensure the compatibility of the two types of the system.

3.4. Four-Variable Model

Four-Variable model [20] (expanded in [8]) is a systematic approach to specify requirements between hardware and software parts. It can be used to better define the boundary between cyber and physical components, and to establish a consistent messaging protocol between the two sides. As shown in Figure 5, there are four variables of interest in this approach.

- MON, the monitored variables. They are parts of the environment (physical components) that the system (cyber components) monitors and may reacts to,

- CON, the controlled variables. They are parts of the environment that the system can influence and control,
- INPUT, the input variables. They are where system accepts input data from the monitored variables, and
- OUTPUT, the output variables. They are where system produces the output to change the controlled variables.

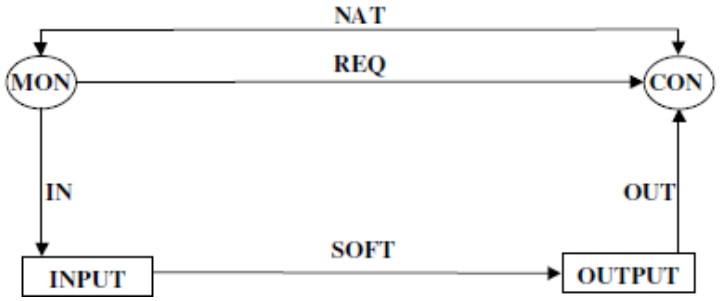

Figure 5. The original Four-Variable model proposed by Parnas and Madey *[8]*

In addition, four relationships are given to portray the way they interact.

- NAT specifies the natural constraints enforced by the environment. They are usually some physical limits that are not possible to overcome,
- REQ is the system requirements that exhibits how controlled variables changes in response to changes in monitored variable,
- IN specifies how monitored variable are translated into input, and
- OUT specifies how output variables are translated into controlled variables.
- SOFT variables are simply the resultant software behavior from the above four relationships.

3.4.1. Analysis of Four-Variable Model.

An interesting feature in the model above is the "overlapping" between NAT and REQ. This can be understood as that the environment has the tendency to behave in a natural way in the absence of perturbance. With the system in place however, some aspects of the environment are no longer dictated by natural forces alone – the system can also add its influence on the controlled variables. Another distinct feature is that all four variables are connected though some intermediate means. This makes the model more rigorous with how variables interact with each other while leaving room for adaption in specific domains. Finally, controlled variables are not allowed to directly interact with input variables, and similarly monitored variables are not to directly influence the output variables. This ensures that data in this system undergoes necessary transformations in the system and it provides maximum separation between the environmental side and the system side.

3.4.2. Adaption of Four-Variable Model to Generic CPS

The Four-Variable model provides a rich yet generic approach that allows for adoption in CPS studies. For example, Figure 6 below (adopted from the study in [21]) shows an application of Four-Variable model in a typical CPS.

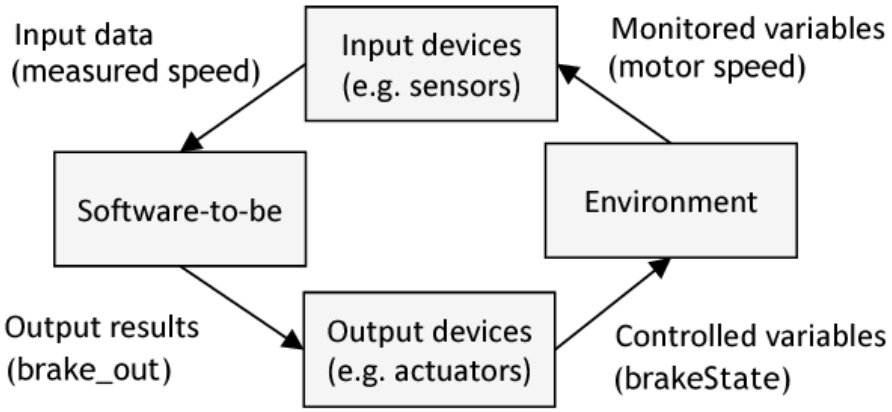

Figure 6. Four-Variable model is used as a guide towards more rigorous and uniform design for CPS *[21]*

It is worth noting that instead of focusing on the four variables, this mapping puts an emphasis on the relationships between the variables. This corresponds to a goal of CPS design and analysis – to better understand and specify the system in question, as well as the environment it is in. The input and output devices, in the case of CPS, can be sensors and actuators. As an example, consider an electric scooter that engages the brake whenever the speed of the scooter exceeds a certain limit, for example 15 mph.

The following sequence of steps can occur:

- The environment (scooter motor) provides some information of interest (e.g., motor speed in rpm) to the sensor (speed sensor).
- The sensor translates the information into a digital format (speed in mph) for the software (speed controller).
- The software then decides an appropriate course of action based on the input received (triggers brake if speed exceeds 15 mph).

- The actuator (brake) acts on the command of the software and possibly change some aspects of the environment (brake slows down the scooter by applying force on the wheel).
- The cycle repeats as the environment provides updated data to the sensor, and software decides on next course of action and may or may not use the actuator.

It is clear that the four variable is well suited for studying CPS. In the following sections, the problem and the approach will be defined in terms of the Four-Variable model input and output data to ensure the material can be well understood and follows good design principles.

4. Related Works

Many approaches have been proposed, developed, and used to co-simulate CPS as combined continuous and discrete time models. These approaches include providing a middle layer that translates the interaction between two different kinds of models and the use of existing standards to accommodate co-simulation to create new frameworks. The following paragraphs examine a selection of representative works on the use of FMI and FMU in view of an approach for hybrid modeling and specific applications.

4.1. Co-simulation of Hardware and Software Using FMI

Masudul H. Quraishi, Hessam S. Sarjoughian, and Soroosh Gholami proposed an approach to enable co-simulation capability for the DEVS-Suite simulator [9]. It focuses on using FMI 1.0 to coordinate parallel DEVS models exemplified for computer hardware with MATLAB algorithms. The overarching architecture of this approach is used in this thesis and shown in Figure 7. This approach is demonstrated for modeling and simulating a DEVS representation of a circular buffer used in Network-on-Chip (NoC). This computer hardware model is based on the Register Transfer Level (RTL) along with its supporting modules. The software for choosing routes in the NoC is a simple two-dimensional algorithm implemented in MATLAB and exported as an FMU. This model serves as the embedded firmware for computing routing path when requested. The two parts are then coupled together with the FMI++ interface [22], which is a C++ wrapper for interfacing with FMUs. This approach shows designing embedded systems

such as single-cycle, multi-cycle, and pipeline processor using RTL-DEVS [23] and basic algorithms.

In the design for the co-simulation of DEVS and FMU, the time allocated for computing the algorithm is instantaneous. The execution of finding a route is untimed, though it is synchronized with each execution time step of the hardware. In other words, each DEVS simulation step can consume logical time, but the FMU for the software does not. Thus, to extend this approach to CPS, it is necessary for both the cyber and physical models to consume time. The interface adapter developed in this approach does not use the FMI functionality where the operations of FMU consume time. The use of FMI++ also presents a layer of complexity to the simulator, due to the use of JNI.

4.2. Hybrid Co-simulation of FMUs in MECSYCO

Benjamin Camus, Virginie Galtier, and Mathieu Caujolle proposed using Multi-agent Environment for Complex System CO-simulation (MECSYCO) platform supported with FMI to co-simulate Cyber-Physical Systems [24]. As a DEVS wrapping platform, MECSYCO supports integrating models defined according to the hybrid DEVS and Differential Equation System Specification (DEV&DESS) formalism [4]. This platform includes a forecasting strategy to look for next event, and an algorithm to locate state changes in FMUs compliant with FMI 2.0. In this approach, multiple DEVS and DESS FMU models are embedded in DEVS framework and connected through *m-agents* and connecting artifacts to enable parallel execution of DEVS models. This integration strategy allows for interaction between FMU and DEVS models, and individual

component can be managed by the DEVS master simulator. This platform also supports distributed and parallel co-simulation.

While this approach provides a generic solution to co-simulating DEVS and DESS models, it requires that FMUs to be connected to the co-simulation by several artifacts that interface with the FMU. These artifacts attempt to discretize the FMU by translating some state changes in FMU to discrete events. Thus, an event-detection function must be implemented to determine if state-events have occurred in the FMU, so that the FMU can be used with DEVS simulators. As the DEVS formalism requires each model to know when the next event shall occur, and the FMU operates like a black box on a time-stepped basis, a bisectional search is needed to determine the time of next state-event. Basically, the algorithm repeatedly simulates and performs single-state rollback on the FMU with an ever-decreasing interval, until the state-event is located.

4.3. Other Related Research

Thierry Nouidui et al. proposed a co-simulation platform *CyDER* that provides an open standard for co-simulating a wide range of power systems [25]. *CyDER* is composed of multiple components, including a number of power grid related FMUs that can be combined to form a power system, and a utility that exports Python functions as FMUs to facilitate control of the system. It also supports real-time simulators exported as FMUs. A master simulator responsible for coordinating the simulation is also proposed. The master simulator solves dependencies between FMU components, provide inputs to the FMUs, and step them together. The output from each step is collected and can be used

for further processing. *CyDER* can be seen as an extension of existing FMI 1.0 and 2.0 standards that targets specifically electrical systems. Some tools were developed based on *CyDER* to support analysis of a power grid. The *CyDER* platform also does not account for DEVS formalism, including its messaging protocol and synchronization, making it not suitable for co-simulation between DEVS models and FMUs.

Virginie Galtier et al. demonstrated a new efficient framework *DACCOSIM* for enabling multi-simulation on FMUs based on FMI 2.0 standard [26]. This framework enables graphical setup of distributed simulation for many FMUs. The framework contains utilities for interacting with FMUs as well as communication between FMUs. The framework can run in distributed environments with parallel simulation of FMUs, which has yielded significant speedup versus traditional monolithic, sequential simulations. It features a simple event detection system that, when a change in output (state-event) is detected in the FMU, it will attempt to localize the time when event takes place by performing FMU rollback and step through the FMU in smaller, user-defined step sizes, until the event can be located. It is apparent that this approach is suitable for managing complex systems that are comprised with FMU modules, but it is not suitable for co-simulation between DEVS and FMUs as time synchronization and message exchange protocol is not defined for them. Similar to the *CyDER* framework, the *DACCOSIM* is primarily designed for use between multiple FMUs, as both platforms do not provide sufficient functionalities to correctly execute models with discrete signals.

It is worth noting that none of the works mentioned above uses the four-variable model for architectural requirement specification, which is one of the main features incorporated in the DEVS-FMI.

5. Approach

There are two main issues with integrating DEVS-Suite and FMUs. First, DEVS-Suite does not interface with FMU natively, thus a solution is needed to enable co-simulation while making it as generic as possible. Second, time coordination between DEVS and FMUs is necessary. Thus, a synchronization protocol that ensures correct input/output exchanges is crucial to the co-simulation. Modeling physical devices for simulation in DEVS and Modelica can be also a matter of interest. The following paragraphs explain the approach in this research to tackle these problems.

5.1. Co-simulation between DEVS and FMUs

Though DEVS-Suite is interoperable with some external simulation tools, such as Simulink, High Level Architecture (HLA), and Simulation as a Service (SimaaS), modification is needed to enable its co-simulation with FMUs. Instead of rewriting the simulation engine and potentially introduce issues into the simulator, Widl and Müller's generic FMI-compliant tool coupling scheme [6] is adopted. The main ideas of this coupling are explained above, and the application of this scheme is fairly straightforward. In DEVS-FMI, the DEVS-Suite simulator is considered the master simulator that handles coordination and messaging between DEVS models and associated FMUs. A separate master simulator is not used as it would be unnecessary and only adds to the complexity of the problem. FMUs are considered slaves as their simulation and scheduling depends on the DEVS-FMI and ultimately DEVS-Suite.

5.1.1. DEVS-Suite and DEVS Model

DEVS-Suite supports modeling and simulating atomic and coupled parallel DEVS models. The execution logic is handled by atomic simulator and coupled simulator. Since all models, including single atomic models, will be eventually simulated via a coupled simulator, it is sufficient to work on only the coupled simulator to enable co-simulation logic for all DEVS models. The modification however is kept to minimum, as additional changes can introduce unintended consequences to the simulator, which is undesired. To ensure maximum compatibility with the existing DEVS-Suite modeling structure, no change is introduced to the DEVS model constructs. In other words, existing models as well as future models can still be used on the simulator without modification.

5.1.2. DEVS-FMI Interface

The DEVS-FMI interface is designed to target FMUs that complies with the FMI version 2.0 using the generic FMI tool coupling design [6]. The updated DEVS-FMI interface follows DEVS-FMI 1.0 which uses similar FMI tool coupling design. To achieve this, three capabilities are required:

- API access (preferably in Java) to FMUs that enables basic interaction with them, such as starting, stopping, performing input, and gathering output.
- Input and output communication with DEVS-Suite simulator.
- Co-simulation logic and controls according to the FMI standard.

The DEVS-FMI interface is used for the input and output devices as specified in the Four-Variable model. The DEVS-FMI is used to integrate FMUs with the DEVS-Suite Simulator. For example, it can obtain outputs from the FMUs and convert them to different data types for use in DEVS-Suite, and vice versa.

There are a number of existing FMI frameworks and libraries that provides functionality needed for interacting with FMUs. For example, FMI++ (C++) [22], JavaFMI (Java) [7], FMPy (Python) [27], the original FMI Library (C) [28], and many more [29] [30] [31]. JavaFMI is chosen for three reasons. 1) It provides a simple and intuitive API for interacting with FMUs, 2) it has been in development since 2013 and still under active development (as of November 2020), and 3) it is written in Java, which means no additional adapters or interfaces are needed to use the library, which improves performance and reduce likelihood of problems.

The second capability can be realized with some modification to the coupled simulator. To enable messaging to the DEVS models, only a reference to the current model is needed, and DEVS-FMI can perform information exchange with the model reference.

The third capability is left to the user. Currently, specific co-simulation logic has to be developed for every combination of DEVS and FMU models. A basic implementation that contains necessary utility functions and a public interface is developed. This interface provides basic support for different use scenarios. It offers a

starting point for developing custom logic needed for co-simulation. The detailed design of the DEVS-FMI extension will be described in next chapter.

5.1.3. FMUs

A FMU for co-simulation is a self-contained unit that can be treated like a black box. Inner workings of FMUs are hidden as the functionality of each FMU is exposed through an interface established by FMI standard. OpenModelica FMUs are taken as given are used to be co-simulated with co-s with DEVS models.

5.2. Timing Consideration

An inherent requirement for co-simulating DEVS models and FMUs is correct coordination of time advance in two different simulators. As explained above, DEVS-simulator may not necessarily use a real-time clock; FMUs for continuous models may also be executed in logical time. In this thesis, the DEVS-FMI is restricted to support logical time co-simulation. The continuous time base is used for DEVS and Modelica models. DEVS models are simulated in discrete time steps while Modelica models are simulated using continuous time in miniscule discrete time steps. Given the discrete time step simulation for the FMU's, DEVS models can have input and output interactions with FMU's at discrete time instances.

5.2.1. Synchronization Protocol

The disparity in timing leads to problems in synchronization. If both parts were real time or discrete, syncing would be relatively simple as timing scheme would be

similar. To enable co-simulation between discrete time and continuous time, the following protocol is established.

1. Initialize the DEVS models.
2. Perform the simulation steps on DEVS models.
    a. If there is an FMU associated with a particular DEVS model:
        i. Initialize the FMU.
            1. If on the first iteration, initialize with predefined initial values.
            2. Otherwise, initialize with states from the previous iteration (step iii) and output from the DEVS model (step vii).
        ii. Perform simulation on the FMU for some amount of time (e.g., 0.5 seconds) and some precision (e.g., 0.2 milliseconds).
        iii. Obtain the output from the FMU by reading state variables from it.
        iv. Process the data and inject them to appropriate ports on the DEVS model.
        v. The input from FMU model, along with other input from the user, is used to perform the step in the DEVS model. Then, the current FMU session is terminated.

vi. DEVS produces some output commands to the FMU,

vii. The output commands are transformed into some input values to the FMU.

b. If there is no FMU associated with the DEVS model, no additional action is taken. DEVS model is simulated as normal.

In this protocol, the FMU receives input from the DEVS-FMI and executes for a prescribed time step. At the end of the time period, the DEVS-FMI sends the FMU's output to the DEVS-Suite simulator which are input events to designated DEVS models. States at the end of the previous simulation are carried to the next simulation step in DEVS. Therefore, this protocol can correspond to the FMU being simulated for one or more discrete time steps (i.e., the DEVS-FMI only receive the output generated at the end of the FMU execution step). Consequently, co-simulation accuracy depends on input and output frequency between DEVS and FMU. Reducing the time period for FMU requires more execution time for co-simulation to complete. Therefore, less frequent input and output can result in inaccurate or wrong simulations.

It is also not feasible to run both the DEVS models and the FMUs concurrently and produce deterministic results for two reasons. 1) The DEVS models are not guaranteed to produce results within any particular time, therefore running the FMU while the DEVS model is producing an output can led to unintended results. 2) Some FMUs, like the ones generated by Modelica, does not respond to inputs after simulation is

started. It is therefore impossible to input commands from the DEVS models to Modelica FMUs in the DEVS FMI approach defined in this thesis.

5.2.2. Discussion on Some Possible Exceptions

In the DEVS-FMI, the correctness of the simulation relies on the assumption that 1) DEVS simulator and FMUs take turns to receive input and produce output and 2) neither side fails to produce output or receive input at scheduled time instances.

The order of execution is controlled by the DEVS-Suite master simulator and defined in the relevant co-simulation module. If programmed correctly, a DEVS model and a FMU will execute alternatively. However, should the DEVS model or the FMU executes out of order (e.g., the DEVS model executes twice before the FMU), the simulation is considered out of sync.

- The DEVS model can execute multiple times during one execution cycle of the FMU. The behavior of the FMU depends solely on the latest input it receives from the DEVS model. Conversely, the behavior of the DEVS model may depend on receiving input from FMU between its internal state changes. In these situations, the co-simulation of the DEVS and Modelica models is not as accurate as they need to be.
- The FMU can execute multiple steps during one execution step of the DEVS model. In this scenario, the accuracy of the DEVS model depends on the FMU's latest output. Since the FMU is terminated and re-initialized with every step of the DEVS model, the FMU uses the latest outputs from

the DEVS model during its simulation steps. The FMU is a black box to the DEVS simulator (i.e., intermediary outputs due to internal execution steps in the FMU are not available to the DEVS simulator).

- The DEVS-FMI can send DEVS model output only at the start of FMU execution. Some FMUs, such as the ones produced by OpenModelica, do not react to the input after the simulation has started. Therefore, the output can be sent only after the FMU has finished its execution. For time-sensitive applications, it is possible to reduce the simulation time that FMU takes each step (e.g., 0.2 seconds) so that the output from the DEVS model can be effective on the FMU sooner.

5.3. Electric Scooter Example

To showcase the capability of DEVS-FMI 2.0, a simple model for select components of an electric scooter is created. This model is intended to recreate a typical rental electric scooter that can be found in major cities. The model is not intended to capture all aspects and properties of a scooter. This model shows the speed of the scooter motor being controlled by a rider's desire to go faster, slower, or stop. The detailed specification of the scooter can be found in APPENDIX I. Only an electric motor, a battery, and input/output devices, are considered to represent the physical components. A basic model imitating a rider making simple changes to the motor speed with braking and a simple model for commanding speed control represent the cyber components.

Therefore, the electric scooter model is divided into two parts. It contains a DEVS coupled model which corresponds to the software controller, and a Modelica model which corresponds to a physical part of the scooter. In addition, co-simulation logic for the electric scooter is provided by extending creating an electric scooter module for DEVS-FMI 2.0.

6. Design

A high-level design of the DEVS-FMI interface with its simulators is shown in Figure 7. This design makes use of the generic FMI-compliant tool coupling scheme proposed by Widl and Müller to establish a lightweight and reliable co-simulation and messaging mechanism between DEVS-Suite and external FMUs.

DEVS-Suite is the simulator responsible for simulating discrete event models, such as the software component of the scooter. DEVS-FMI interface situates inside the DEVS-Suite and interfaces with the external FMUs via JavaFMI library. Modelica, on the other hand, generates ScooterMotor FMU that contains select physical component for co-simulation. In this architecture, DEVS-Suite is the master simulator that manages the data flow and synchronizes the execution of the DEVS models and FMUs. The following paragraphs explains the construct of individual components in more detail.

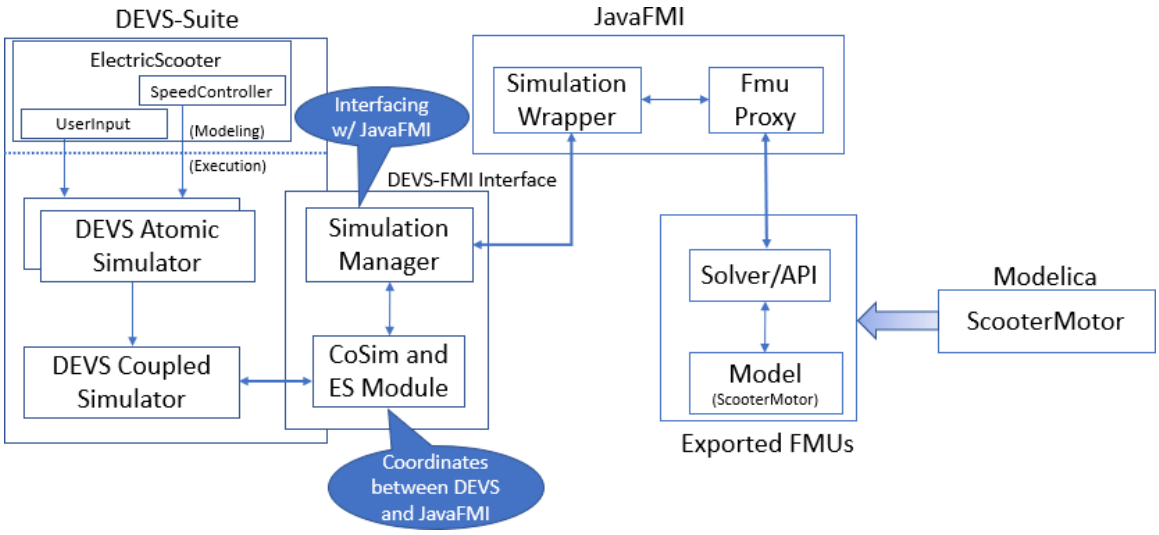

Figure 7. High-level overview of the DEVS-FMI design for DEVS-Suite simulator

6.1. DEVS Electric Scooter Model

The electric scooter coupled model is made up of two constituent atomic models: SpeedController, which controls the speed of the motor using user input and data from the motor; UserInput, which simulates user interaction with the scooter using some predefined input sequence. Together, they can simulate an electric scooter system that is controlled by a rider.

6.1.1. DEVS Atomic Model

To better understand the design of the electric scooter DEVS model, it is necessary to look at the basic atomic model construct in DEVS-Suite. Figure 8 shows a class diagram that contains the atomic class and the devs abstract class it inherits from. Some less relevant methods and attributes are omitted for clarity of the diagram.

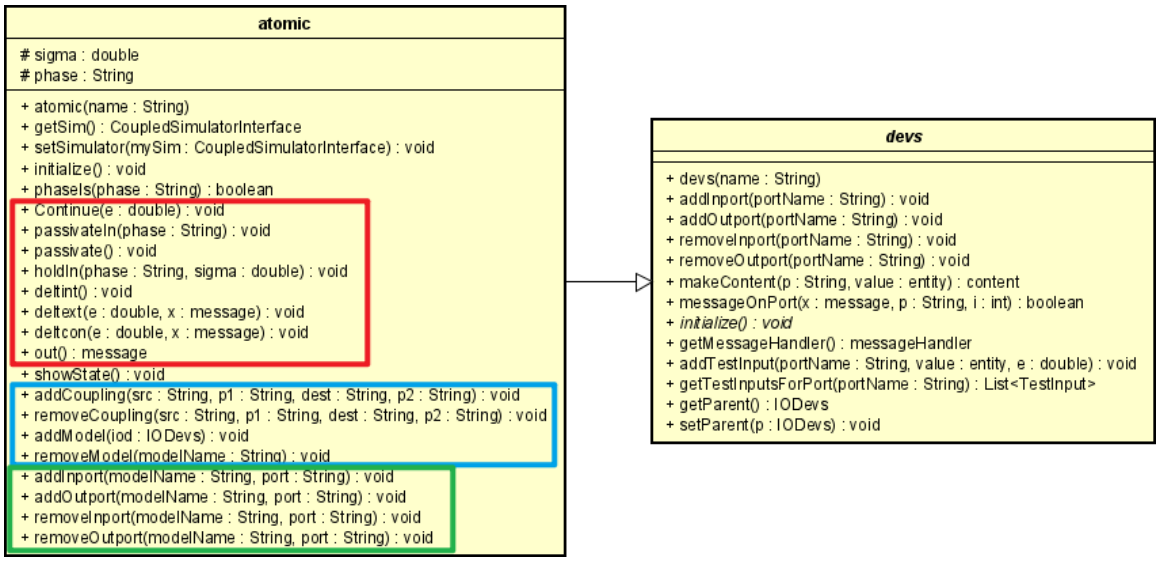

Figure 8. Class diagram of atomic model and devs abstract class in DEVS-Suite. Some methods and attributes are omitted for clarity

Three important groups of functions are outlined above.

- The red box contains functions that implements the basic functions of the parallel DEVS atomic model, such as internal and external transitions. Collectively they can specify the generic behavior of any model. deltint(), deltext(), deltcon(), and out() functions are to be expanded by the model to define its behavior on internal transition, external transition, confluence, and output functions, respectively.
- The blue box contains functions for defining variable structure models.
- The green box contains methods to add and remove input and output ports on the model.

6.1.2. Electric Scooter Model - Structure

Electric scooter model extends upon the basic atomic model and its class diagram is shown in Figure 9. Note the added attributes and states in both models. It is apparent that the ElectricScooter is a composition of its constituent models, SpeedController and UserInput. The ScooterConstruct() function is called when the ElectricScooter coupled model is created, in which the speed controller and user input models are added and coupled together. Input and output ports are added as well.

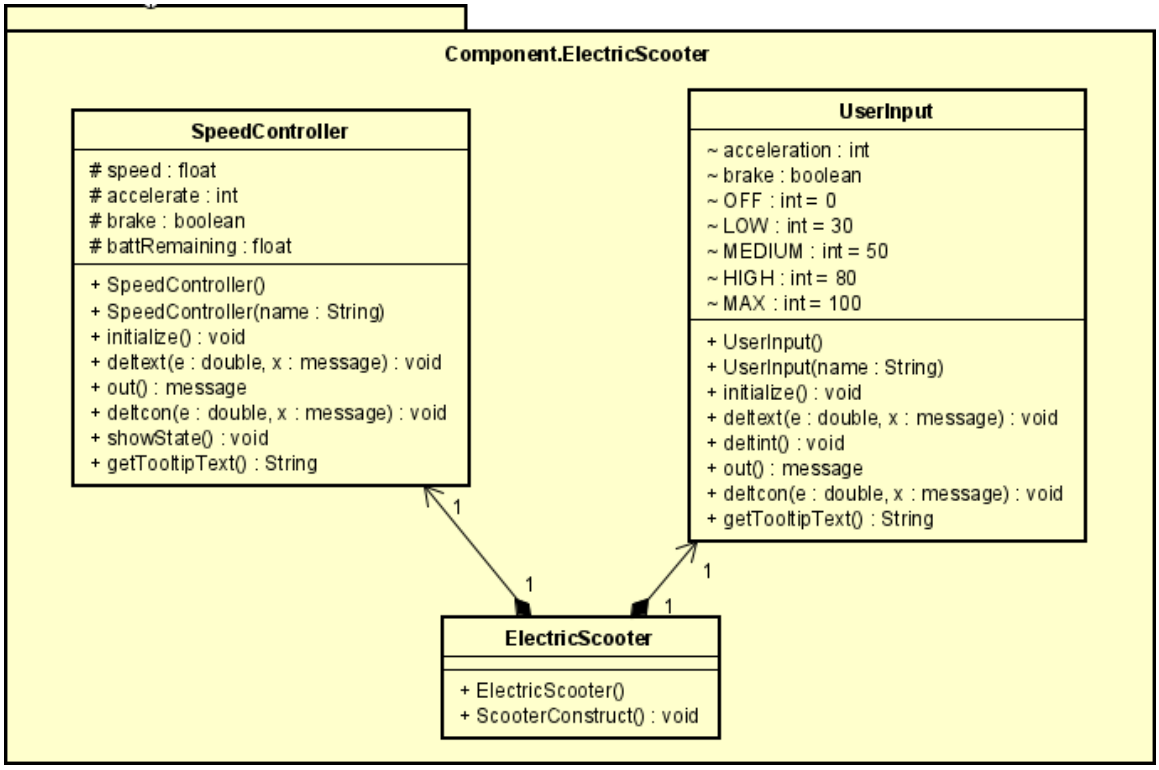

Figure 9. Class diagram for the electric scooter component (package)

The corresponding coupled model in DEVS-Suite component view is shown in Figure 10. This visual representation supports animation of state changes in the atomic models and sending/receiving of messages. <Controller> corresponds to the SpeedController class and <Input> corresponds to UserInput class. The ports on the models and their uses are explained below.

- ElectricScooter
    - Input ports
        - start: accepts any input and will relay the signal to <Input>. Doing so will start the execution of the scooter.

        - stop: accepts any input and will relay the signal to <Input>. Doing so will stop the execution of the scooter.
    - Output ports
        - brake: sends a brake message produced by <Controller>. Possible values are True and False.
        - power: sends a power message produced by <Controller>. Possible values range from 0.0 to 100.0.
- <Input>
    - Input ports
        - start: accepts any input. Upon receiving an input, it will start producing outputs (acceleration and/or brake) according to some predefined pattern.
        - stop: accepts any input. Upon receiving an input, it will stop the production of output.
    - Output ports
        - acceleration: indicates how much the user has pressed the acceleration handle. Possible values range from 0 to 100.
        - brake: indicates whether the user has pressed the brake handle. Possible values are True and False.

- <Controller>
    - Input ports
        - accel: accepts input from <Input>, this indicates how much the user is pressing the acceleration handle and controls how much the scooter should be accelerating (providing no other factors are preventing acceleration). Possible values range from 0 to 100.
        - batt_remaining: accepts input from the FMU, this indicates the percentage of battery remaining in the system. Note the lack of connection from upstream – this is intended and will be explained below. Possible values range from 0.0 to 100.0.
        - brk_toggle: accepts input from the <Input>, this indicates whether user is engaging the brake and controls if the brake should be applied. Possible values are True and False.
        - sc_speed: accepts input from the FMU, this indicates the speed at which the scooter is travelling. Note the lack of connection from upstream – this is intended and will be explained below. Possible values range from 0.0 to (Infinity), though realistically it is unlikely to see numbers above 25.0.

- Output ports
  - brk_out: indicates whether the brake should be applied. Possible values are True and False.
  - pwr_out: indicates how much power should be applied to the motor. Possible values range from 0 to 100.
- The inputs for the batt_remaining and sc_speed are provided by the ScooterMotor FMU. The co-simulation design requires fetching these inputs from the FMU and injecting them into their respective input ports. According to the synchronization protocol in Timing , the FMUs associated with the DEVS models are executed first. Some outputs from these FMUs are used as input to the DEVS model. In the case of the electric scooter example, these two ports are getting updated inputs from the electric motor FMU in each round.

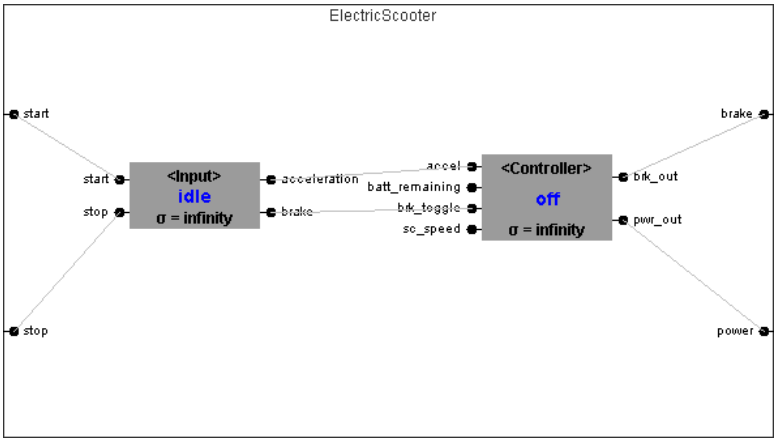

Figure 10. Electric Scooter visual representation in DEV-Suite, with couplings between them shown

6.1.3. Electric Scooter Model - Logic

The logic for the speed controller is defined using the following rules, and the rule higher up takes precedence when multiple rules can be applied at the same time.

1. When user presses the brake handle, the controller enables the brake and reduce power output to zero.
2. When remaining battery has less than 10% capacity, enter the power-saving mode and ignore acceleration input.
3. When speed of the scooter is over some predefined limit (e.g., 15 mph), reduce power output to zero.
4. When user presses the acceleration handle
    a. If the speed is already over the predefined limit, not action is taken.
    b. If the speed is below the limit, increase the power output proportional to how much the handle is depressed.
5. When no input is received, take no action, and wait for next input.

More formally, the behavior of the speed controller can be defined in the advanced state machine diagram, shown in Figure 11. Colors are added to some transitions for clarity: blue transitions indicate they are transitioning into Idling state, while red ones indicate they are transitioning out of Idling state.

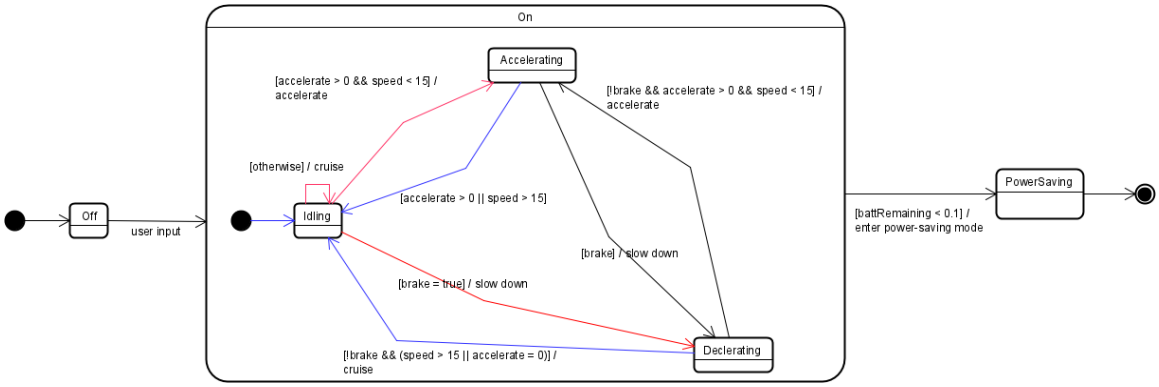

Figure 11. Advanced state machine diagram for speed controller

On the other hand, user input contains predefined sequence of input to simulate a particular user's interaction with the scooter. The following sequence depicts one of the possible profiles.

1. User accelerates at LOW (30) rate.
2. User accelerates at MEDIUM (50) rate.
3. User accelerates at MAX (100) rate.
4. User let the scooter coast (OFF).
5. User brakes.

Other profiles can be easily assembled by rearranging the order of operations or adding and removing operations.

6.2. Modelica Electric Scooter Model

The Modelica counterpart of the electric scooter model contains a direct current permanent magnet (DCPM) motor, electrical components around the motor, and a simplistic battery component. DCPM motors work similar to the ones installed in a real-life scooter. Figure 12 shows these components and connections between them in

OpenModelica. Setup of some important parameters in the model is listed in APPENDIX II.

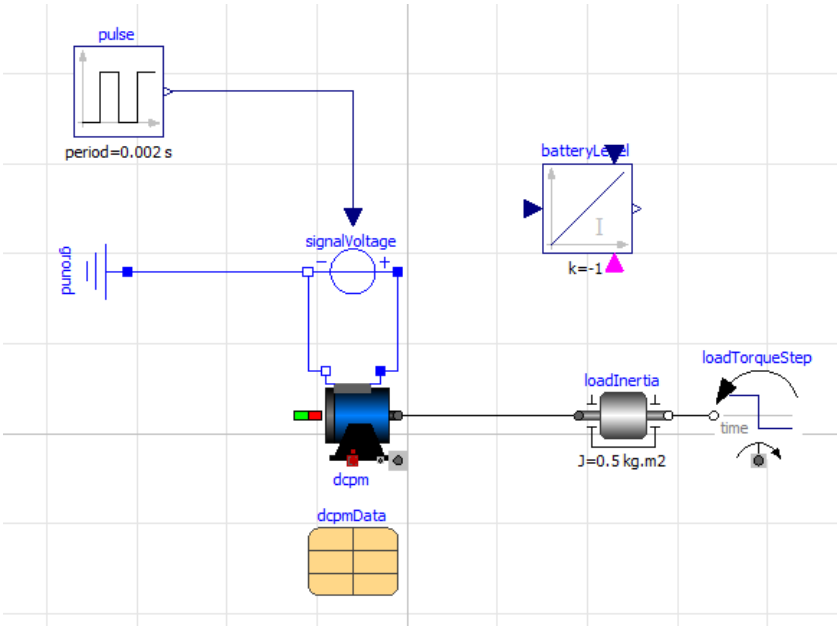

Figure 12. Part of the electrical scooter modeled in Modelica showing various components. Adopted and expanded from the built-in DCPM example

DCPM is the central piece in the Modelica model and its behavior directly impacts the behavior of the scooter system. The underlying logic in DCPM consists of a significant number of constituent models and equations, which will not be shown here for simplicity. In essence, the speed of the motor responds to the input voltage – the higher the input voltage is, the faster the motor spins, and therefore faster the scooter goes. In the case of electric scooter, control of the voltage is usually achieved via a technique known as pulse-width modulation (PWM). Instead of directly controlling the input voltage directly, it controls how much time the voltage is on high versus the time on low. The effective voltage is then determined by duty cycle, or the percentage that voltage

signal is on high. In the Modelica model, PWM scheme is implemented using the pulse component, which sends a pulse signal with customizable amplitude, width (duty cycle), and period.

The battery model (batteryLevel) is trivial – it is a simple integrator that has a start value and a slope. As the motor draws power to operate, the remaining value of the battery decreases. The exact slope at any given time is the negative of voltage times the current, which would be the wattage of the motor. This relationship is not shown in the diagram above as the linkage in OpenModelica has different meanings. The dcpmData component contains specific parameters to control the behavior of the electric motor, such as its nominal voltage, nominal current, nominal speed, nominal resistances, and many more. The loadInertia and loadTorqueStep parameters provide a load (resistance) profile on the motor to simulate physical loads on the scooter, such as a rider.

6.3. DEVS-FMI Adapter

The DEVS-FMI adapter is an implementation of the adapter in the generic FMI-compliance tool coupling scheme. It is an extension to DEVS-Suite simulator that enables co-simulation between DEVS models and FMUs.

6.3.1. Structure and Functionality of the DEVS-FMI Adapter

Figure 13 depicts the DEVS-FMI adapter (FmiCoSimV2 package) as well as some related classes, such as the DEVS coupled simulator and the JavaFMI Simulation class.

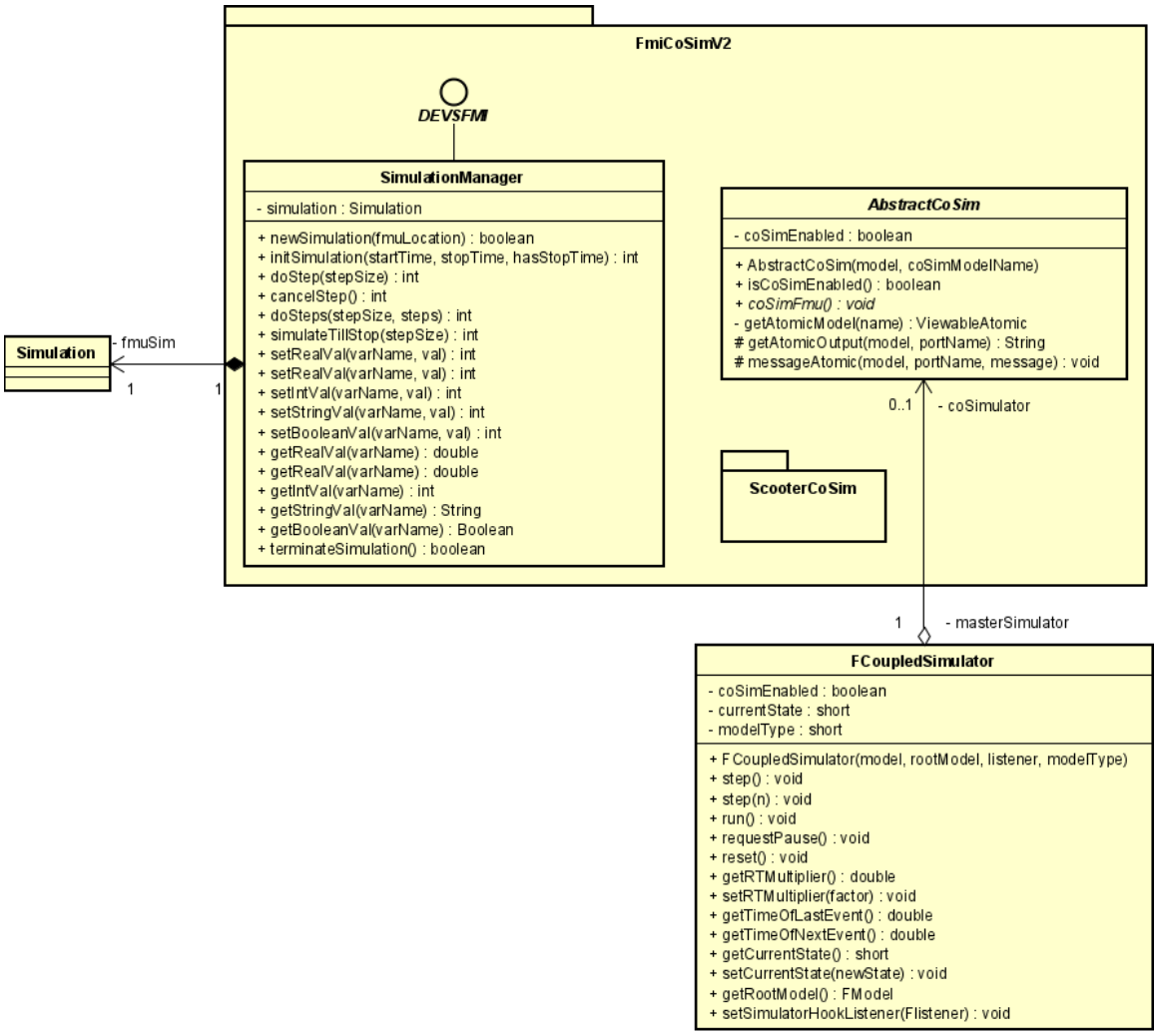

Figure 13. Class diagram of the DEVS-FMI adapter along with related classes

The following functions are implemented in the DEVS-FMI adapter for one or more FMUs. The SimulationManager and AbstractCoSim classes provide sufficient coverage for co-simulating DEVS models and FMUs.

- Interact with FMUs with SimulationManager class. This class makes use of JavaFMI library and exposes relevant functions to facilitate co-simulation. SimulationManager allows for
    - Starting, initializing, terminating the simulation,
    - Stepping the simulation with user-defined step size,
    - Cancelling the step,
    - Stepping the simulation till it stops,
    - Setting real, integer, string, and boolean values on the FMUs,
    - Retrieving real, integer, string, and boolean values from the FMUs.
- Provide a foundation for implementing custom co-simulation layer for any DEVS models and FMUs. One can simply extend the AbstractCoSim class and provide additional logic for co-simulation. The newly extended CoSim modules can be easily integrated with the DEVS models without having to change core simulator logic or DEVS models. The following features are supported in the AbstractCoSim:
    - Initialize the CoSim module and its integration with DEVS-Suite simulator,
    - Get reference to DEVS atomic model by name (to enable interacting with it),
    - Get output on a specific port of an atomic model,
    - Provides input to a specific port of an atomic model.

6.3.2. Electric Scooter Module in the DEVS-FMI Adapter

A module specifically designed for the electric scooter is implemented. It consists of two classes – ScooterCoSim, which extends the AbstractCoSim and includes co-simulation logic for the electric scooter; FmuParam, which is an enum that facilitates data exchange between DEVS-FMI and the electric scooter FMU. Figure 14 is the class diagram for a DEVS-FMI electric scooter module.

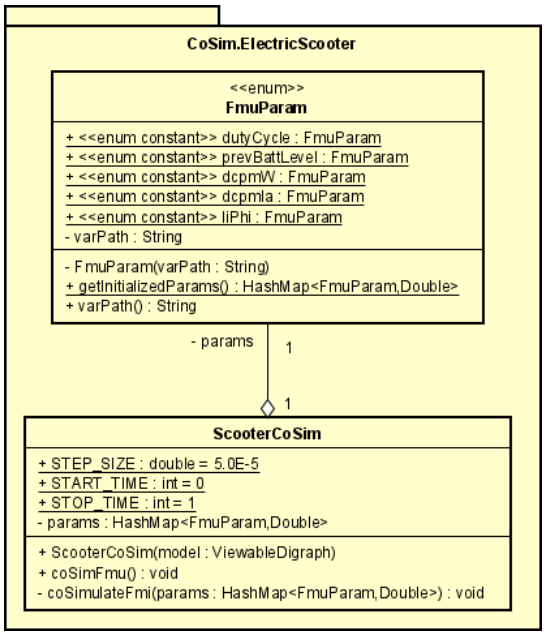

Figure 14. Class diagram of electric scooter DEVS-FMI module

ScooterCoSim contains a public function coSimFmu() that is exposed to the outside. Invoking this function would obtain specific outputs from a given DEVS model, initiate a new round of simulation in the FMU, pass the output from the DEVS model and saved states into the FMU, execute the simulation for some length (e.g., 1 second), and inject the output from the FMU back into the DEVS model for the next step in DEVS.

coSimulateFmi() is a helper function that start a new simulation in FMU and initialize it with a number of variables, including saved states from previous stimulation and output from the DEVS model.

There are 5 parameters that are passed into and out from the FMU in a simulation run. These variables are listed below (all of them are double-typed).

- dutyCycle: Duty cycle for next simulation step that determines the effective voltage to the motor. Ranges from 0.0 to 100.0. This corresponds to pulse.width variable in FMU.
- prevBattLevel: Battery level at the termination of previous simulation, used to initialize the battery level on next simulation run. Ranges from 0.0 to 777,600.0. This corresponds to batteryLevel.y variable in FMU.
- dcpmW: Speed of the motor at the termination of previous simulation, used to initialize the motor speed on next simulation run. Ranges from 0.0 to approximately 2,600.0. This corresponds to dcpm.wMechanical variable in FMU.
- dcpmIa: Electric current in the motor armature at the termination of previous simulation, used to initialize the motor armature current on next simulation run. Ranges from 0.0 to approximately 13.5. This corresponds to dcpm.ia variable in FMU.
- liPhi: Initial rotation angle of the load inertia at the termination of previous simulation, used to initialize the initial rotation angle of load inertia on

next simulation run. Ranges from 0.0 to (infinity). This corresponds to loadInertia.phi variable in FMU.

It is apparent that managing the five parameters can be complex and error prone. Misspelling of the variable name or using a wrong variable can lead to undesired simulation outcomes. To ensure the consistency of parameters in the implementation and eliminate possible mistakes, an enum FmuParam is introduced. It has one type that corresponds to each parameter, and each type also contains a variable varPath that corresponds to its variable path in FMU. This streamlines the access to the scooter FMU and makes the implementation easier to understand. It is recommended that similar enums to be created for other FMUs, along with their co-simulation modules.

The ScooterCoSim performs the following operations for every simulation step.

1. Obtains the pwr_out and brk_state output from the DEVS simulator.
2. Updates the saved dutyCycle state with pwr_out value, and brakeState with brk_state value.
3. Performs co-simulation.
    a. Creates a new FMU simulation instance.
    b. Initializes the FMU with the saved states.
    c. Performs simulation for the predefined length (e.g., 1.0 seconds).
    d. Saves selected state values (prevBattLevel, dcpmW, dcpmIa, and liPhi) from the FMU and terminates the simulation.

4. Convert some outputs from the FMU to a more useful representation.
    a. dcpmW in rad/s is converted into mph (assuming 1.5-inch wheel diameter).
    b. prevBattLevel is converted into battery percentage.
5. Sends updated sc_speed, batt_remaining and brk_toggle as input to the DEVS simulator.

6.3.3. Four-Variable Model Input and Output

As indicated above, the DEVS-FMI adapter is used for the input and output devices defined in the Four-Variable model. For real-world devices, input and output devices translate the analog signals to digital signals from the physical part to the cyber part. Likewise, they also translate digital signals to analog signals from the cyber part to the physical part. Therefore, the DEVS-FMI adapter can be considered as the input and output devices in the co-simulation: the FMU's motor speed and battery level data are provided as primitive data types converted to entity types that can be accepted as input to DEVS models. Conversely, the DEVS ElectricScooter model's power and break outputs are converted to primitive data types supported in JavaFMI (see Figure 10). These devices are implemented in the ScooterCoSim module.

6.3.4. The Big Picture

Figure 15 shows a simplified view of the entire DEVS-FMI adapter, electric scooter models, as well as its related packages and classes. Attributes and operations are omitted for simplicity. Details of each classifiers can be found in above figures.

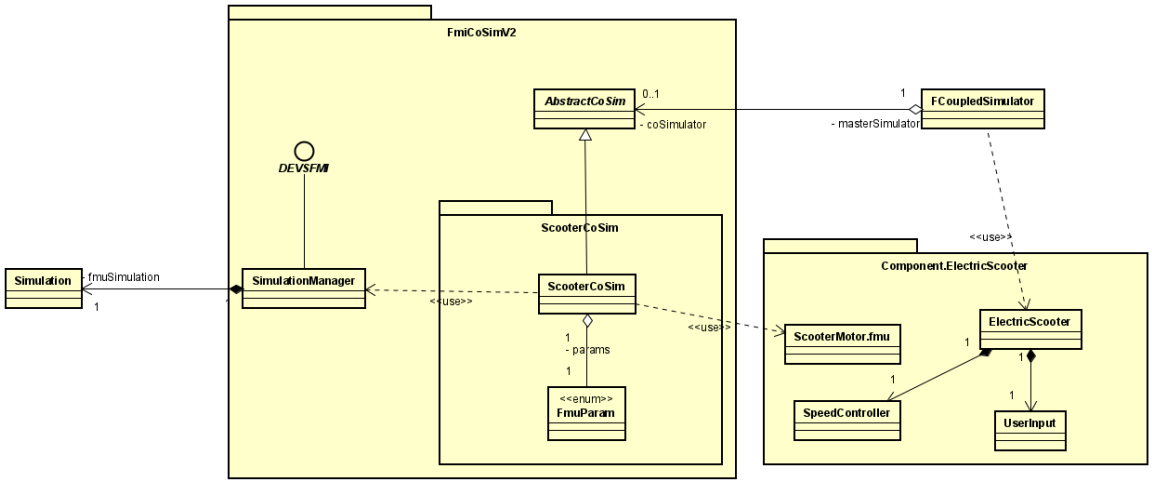

Figure 15. Simplified class diagram for DEVS-FMI interface along with electric scooter models and related classes

7. Experiments

To show the DEVS-FMI is correctly implemented and measure the computation cost for the co-simulation of DEVS models and FMUs, some experiments are developed. The computation cost refers to the physical time, measured using the JVM clock, for the DEVS-FMI to be executed. They show two basic use case scenarios for the integrated DEVS and Modelica models of the electric scooter. The hardware and software configuration of the DEVS-Suite simulator is listed in APPENDIX III.

7.1. Physical Components of the Electric Scooter

To better understand the behavior of Modelica electric scooter model and verify it reflects the characteristics of a real-life scooter well, a series of simulations are performed for the electric scooter motor and battery models using OpenModelica. These simulations have identical setup and initial parameters except for dutyCycle parameter, which portraits the behavior of the scooter model under different user input. Otherwise, the following simulation parameters are used in OpenModelica.

- Start time: 0 seconds,
- Stop time: 20 seconds,
- Number of pulses: 500 (period for each pulse is 0.04 seconds),
- Integration method: dassl,
- Tolerance: 1.0e-6,
- Jacobian: (None).

Figure 16, Figure 17, and Figure 18 show three aspects of interest on the electric scooter motor in different duty cycles. On the first two figures, results are arranged from high duty cycle to low duty cycle from top to bottom. On the last figures, results are from low duty cycle to high duty cycle.

7.1.1. Induced Armature Voltage

Induced armature voltage is the electric potential created by the magnetic field in the motor when it is energized. It is therefore the effective electric voltage in the motor. As shown in Figure 16, the effective voltage responds to the duty cycle parameter in somewhat linear fashion – for example, on 100% duty cycle, the effective voltage appears to be converging to 24 volts, which is the nominal voltage of the motor. On 50% duty cycle, the effective voltage is converging to 12 volts, and so on. It may be unusual that there's negative voltage under 0% duty cycle, but it reveals an important aspect of the simulation setup – that the load acts as a constant torque against motor's direction of spinning. Therefore, the motor instead of converting electricity to rotation of rotor, it spins the rotor and generates electricity. This shows the resiliency in the model setup, as it closely mirrors the behavior of its real-life counterpart. If the load were to be removed, simulation shows the states of the motor remains constant (0 volts) in 0% duty cycle.

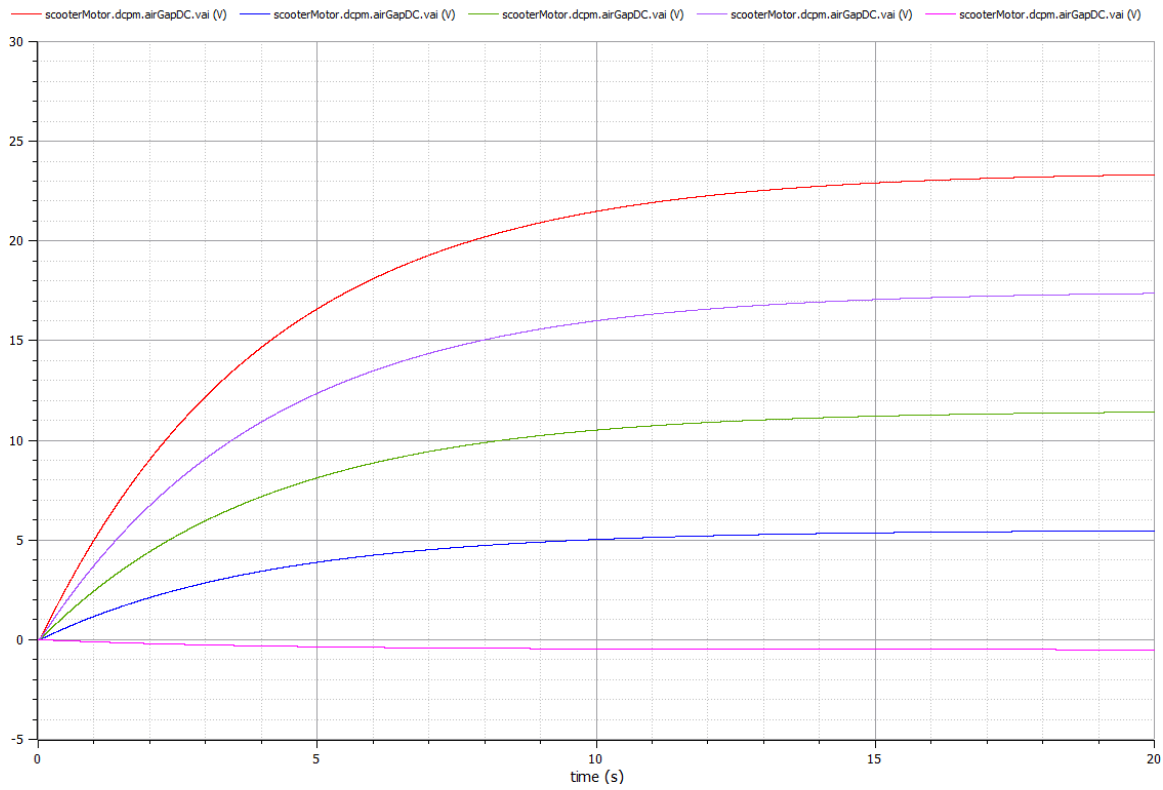

Figure 16. Effective armature voltage of the scooter at 0%, 25%, 50%, 75%, and 100% duty cycle

7.1.2. Speed of the Motor

As shown in Figure 17, the speed of the motor follows closely with the effective armature voltage. This behavior confirms that the speed of motor is directly proportional to the effective armature voltage. Therefore, it is possible to control the speed of the motor (hence speed of the scooter) by controlling just the duty cycle parameter.

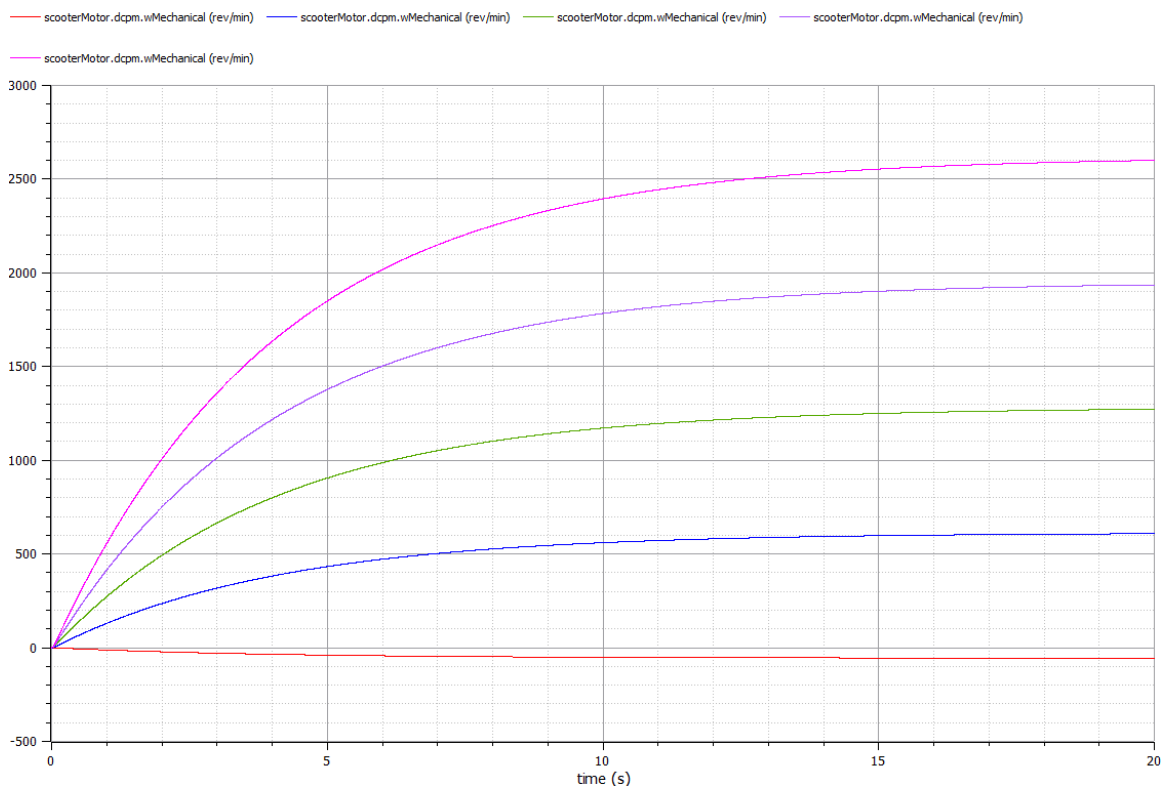

Figure 17. Speed of the motor at 0%, 25%, 50%, 75%, and 100% duty cycle

7.1.3. Battery Level

It is apparent that the remaining battery level also follows closely with voltage level. At 0% duty cycle, a small baseline electricity usage is incurring, which is reflected on the diagram below. As duty cycle increases, the rate of consumption increases as well, resulting in faster decrement of the battery level.

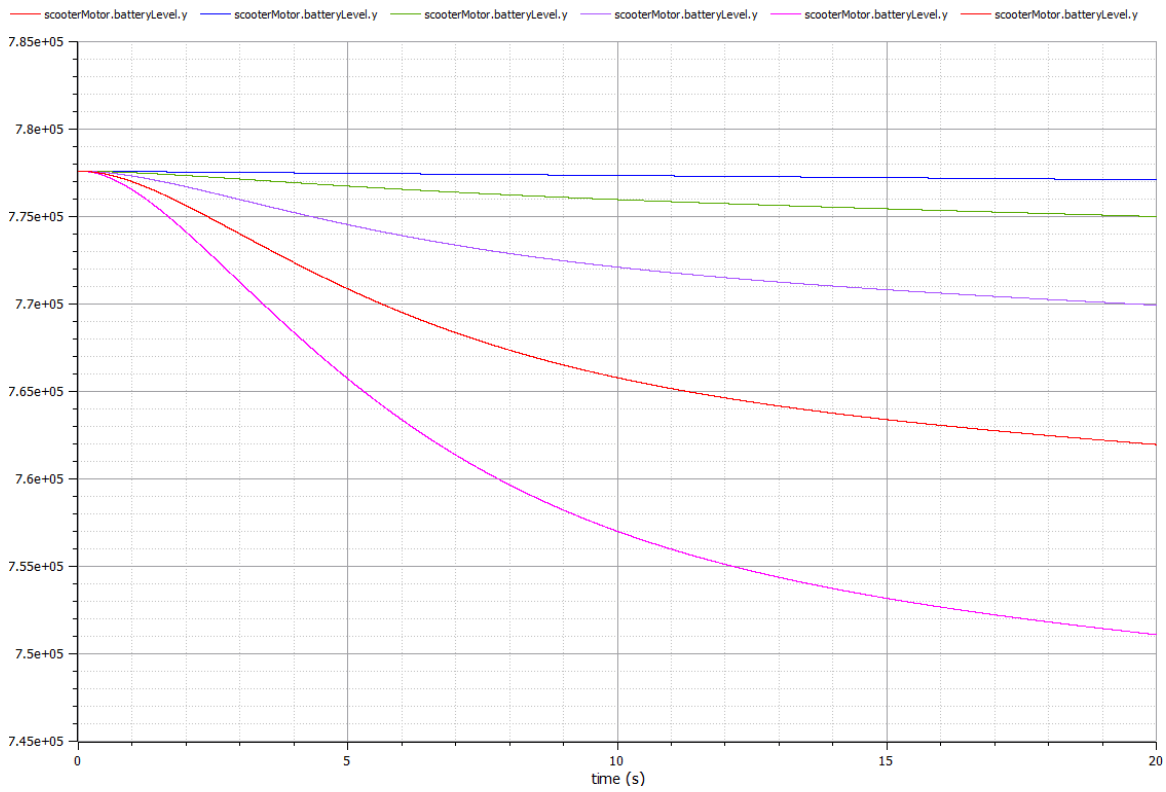

Figure 18. Usage of battery (remaining battery level) at 0%, 25%, 50%, 75%, and 100% duty cycle

These experiments show the choice of the Modelica model for the physical part of the electric scooter shown in Figure 12 is satisfactory.

7.2. Continuity and Accuracy of FMU Simulations

The DEVS-FMI operates on the assumption that FMUs are able to perform accurately to a limited degree with JavaFMI. In addition, terminating and restarting FMU simulations may affect the accuracy of the FMU simulation. The following experiment is conducted to verify these assumptions.

In this experiment, three simulation scenarios based on an identical Modelica electric scooter model are executed. The first simulation is executed on OpenModelica

with the original model construct. Because of this, the output given by OpenModelica (shown in results in page 100 and 102) is considered as the “reference” for other simulations. The other two simulations are based on the FMUs exported from OpenModelica that are simulated in JavaFMI – one that does not stop and restart (similar to OpenModelica simulation), and the other employs stop and restart strategy (similar to DEVS-FMI approach). Otherwise, all three simulations follow the same setup:

- Total execution time is 5 seconds (start time = 0, stop time = 5),
- Step size is 0.00001 (1.0e-5) seconds,
- Identical initial values are used to initialize the simulation,
- Output is sampled once per 0.2 seconds.

To achieve this setup with DEVS-FMI approach, the length of each simulation run is set to 0.2 seconds, and the dutyCycle parameter is to remain constant at 50.0 to simplify comparison.

The initial parameters are configured as following to initiate simulation:

- dutyCycle: 50.0,
- prevBattLevel: 777,600.0,
- dcpmW: 0.0,
- dcpmIa: 0.0,
- liPhi: 0.0,

To track the accuracy of different simulation setups, the outputs of both JavaFMI simulations are compared against the "reference" output from OpenModelica. Specifically, the speed of motor, dcpm.wMechanic is used for comparison.

As shown in Figure 19, the results are nearly identical in these setups. Both FMU simulation methods yields the same results, and they differ from the "reference" values by an average of 1.04% (detailed test results can be found in APPENDIX IV). Thus, the JavaFMI library is able to execute FMUs with acceptable accuracy, given a suitable step size is chosen (more on step size in Impact of Step Size section). The experiment also proves that, if correct states are used, stopping and restarting simulation does not have an impact on the results of simulation.

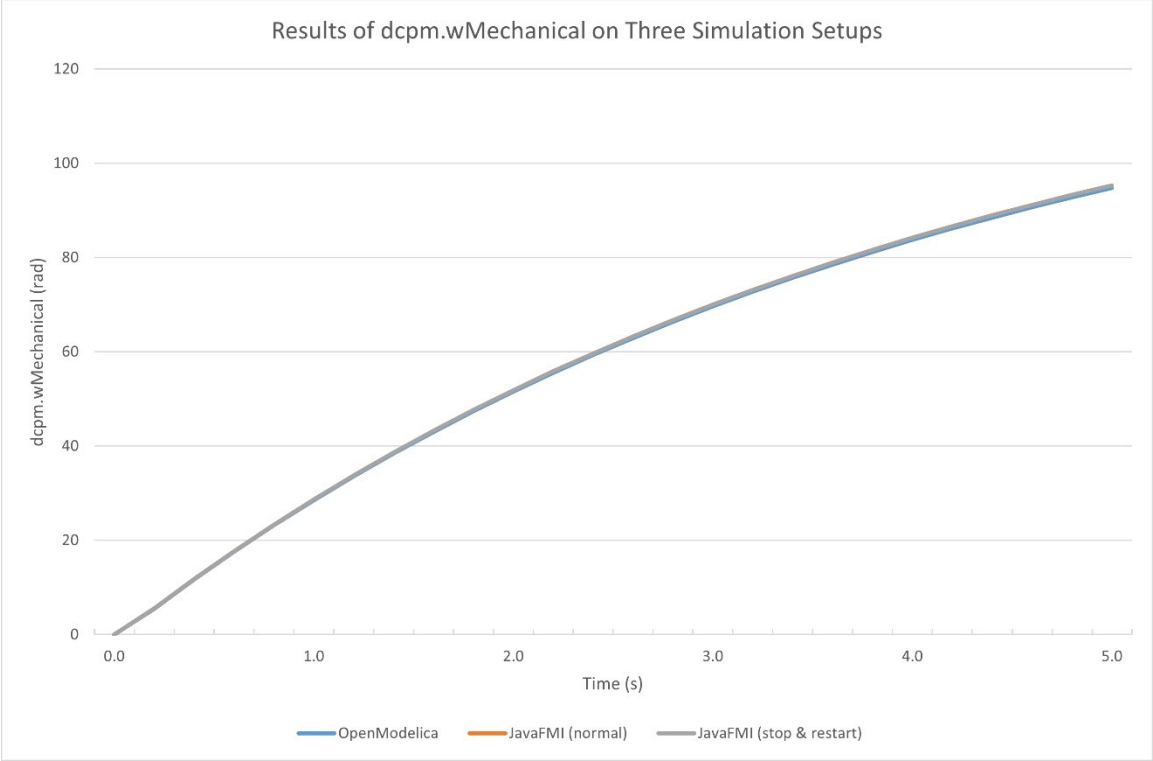

Figure 19. Results of dcpm.wMechanical on three simulation setups. The three curves are almost overlapping, showing that the error of the executions in JavaFMI is minimal

7.3. Impact of Step Size

Step size is a crucial parameter that JavaFMI uses to configure the solver in FMUs. To further highlight the importance of choosing an adequate step size in JavaFMI with exported FMUs, a number of simulations with different step sizes are ran on the electric scooter model and the results are compared. The setup of these simulations is identical to the experiment above, except that they have different step sizes. Particularly, the following step sizes are used for comparison.

- 0.000001 seconds (1.0e-6)
- 0.000005 seconds (5.0e-6)

- 0.00001 seconds (1.0e-5 / DEVS-FMI default)
- 0.00005 seconds (5.0e-5)
- 0.0001 seconds (1.0e-4)
- 0.0005 seconds (5.0e-4)

The results are sampled every 0.2 seconds, and each simulation is run for 5 seconds. The outputs from these simulation runs are compared against the output of the "reference" model configuration from OpenModelica. In addition, these simulations are timed to compare approximate computational cost of the respective step sizes.

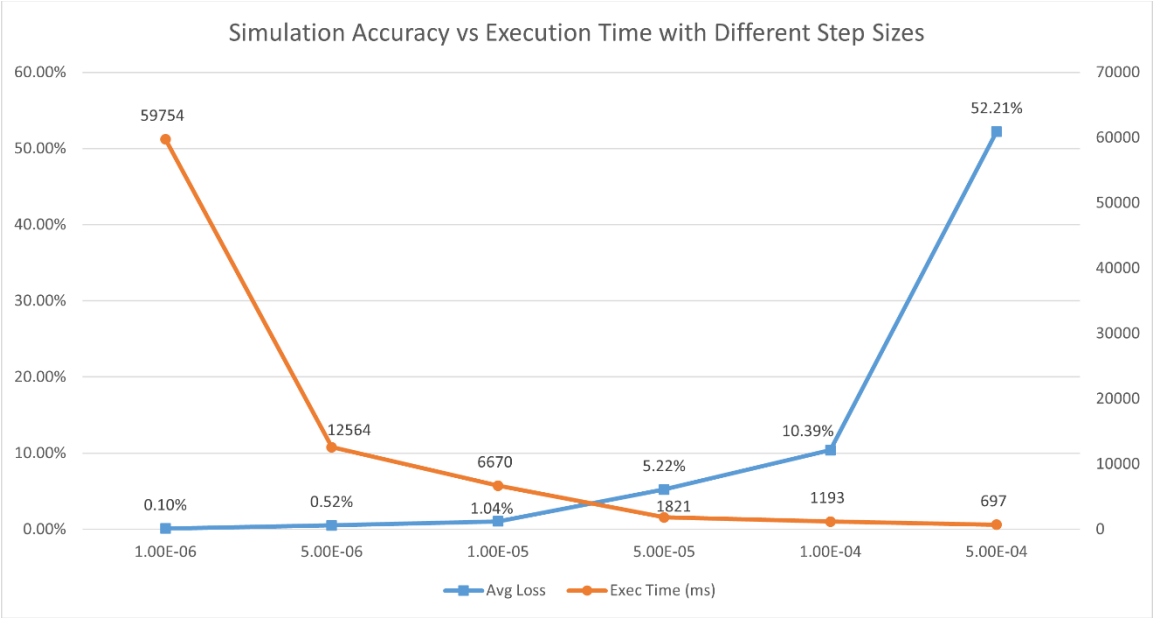

Figure 20. Simulation accuracy of the electric scooter model vs execution time on different step sizes

Preliminary results are shown in Figure 20 and they are observed from single simulation runs. The full results are given in APPENDIX V. It is apparent that as step size decreases, simulation accuracy improves. As expected, decrease in step size results in increase in execution time. Further inspection shows that the relationship between step

size and average loss (error) is approximately linear, while the relationship between step size and execution time is approximately linear as well.

For practical purposes, neither extremes are desirable – if execution time is too long, it can significantly delay the overall co-simulation; if error rate is too high, the simulation results are meaningless. Therefore, it is important to choose an adequate step size that provides sufficient accuracy while bearing possible performance impact in mind. For electric scooter co-simulation, 1.00e-5 seconds is deemed a reasonable time step as it reaches a good compromise between accuracy and execution time.

7.4. Interaction between DEVS Models and FMUs

7.4.1. Experiment 1: A Typical User Input Profile

It is imperative to confirm that the DEVS Models and FMUs are able to communicate correctly via the DEVS-FMI adapter. For this purpose, the electric scooter model described in previous sections is co-simulated as a whole. Specifically,

- The DEVS model is described in the DEVS Electric Scooter Model section,
- The Modelica model is described in the Modelica Electric Scooter Model section.

The DEVS model is simulated in DEVS-Suite simulator with the following user input profile.

Second 1. User accelerates at LOW (30) rate.

Second 2. User accelerates at MEDIUM (50) rate.

Second 3. User accelerates at MEDIUM (50) rate.

Second 4. User accelerates at HIGH (80) rate.

Second 5. User accelerates at HIGH (80) rate.

Second 6. User accelerates at MAX (100) rate.

Second 7. User accelerates at MAX (100) rate.

Second 8. User accelerates at MAX (100) rate.

Second 9. User let the scooter coast (OFF).

Second 10. User brakes.

Second 11. User brakes.

Second 12. User exits the scooter (idle).

The Modelica model is exported as a FMU using OpenModelica built-in converter. The following configurations are used for co-simulation.

- Step size: 0.00001 (1.0e-5) seconds,
- Duration of each simulation: 1 second,
- Start time: 0 second,
- Stop time: 1 second,
- Assumes the diameter of the wheel is 1.5 inches for the purpose of calculating the speed of the scooter.

The FMU is initialized with the following parameters.

- dutyCycle: 0.0
- prevBattLevel: 777600.0

- dcpmW: 0.0
- dcpmIa: 0.0
- liPhi: 0.0

During the simulation, dutyCycle will change according to user input (i.e., how much user depresses the accelerator). The remaining four parameters are simply carried over from one simulation run to next.

Figure 21 shows the DEVS-Suite simulator graphical interface at a time instance during the model execution. Note that DEVS-FMI is enabled, but the graphical interface is not yet updated to reflect the DEVS-FMI integration nor the input coming from FMU.

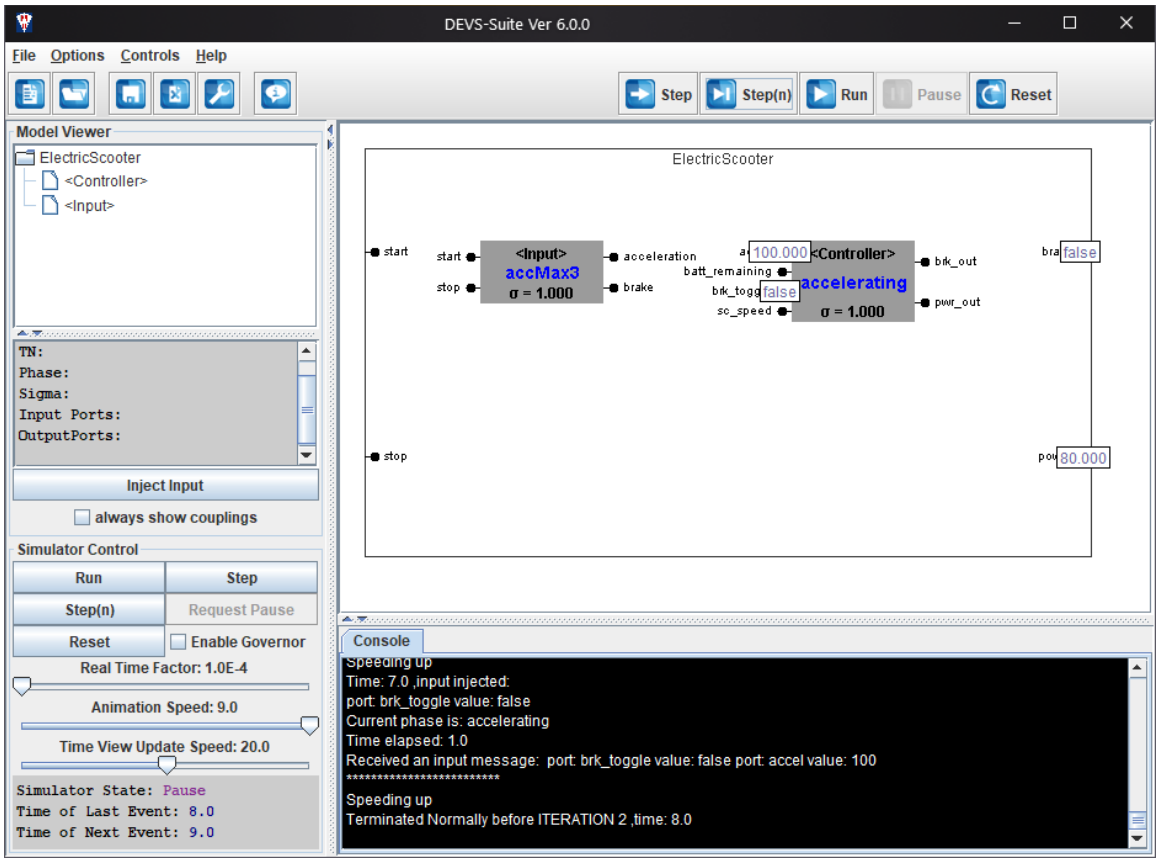

Figure 21. DEVS-Suite interface while executing the electric scooter co-simulation model

Figure 22 depicts partial (first 10 steps) trajectories of break output and power output of the controller model. It also shows a trajectory for phase changes as input from the input module and FMU is received.

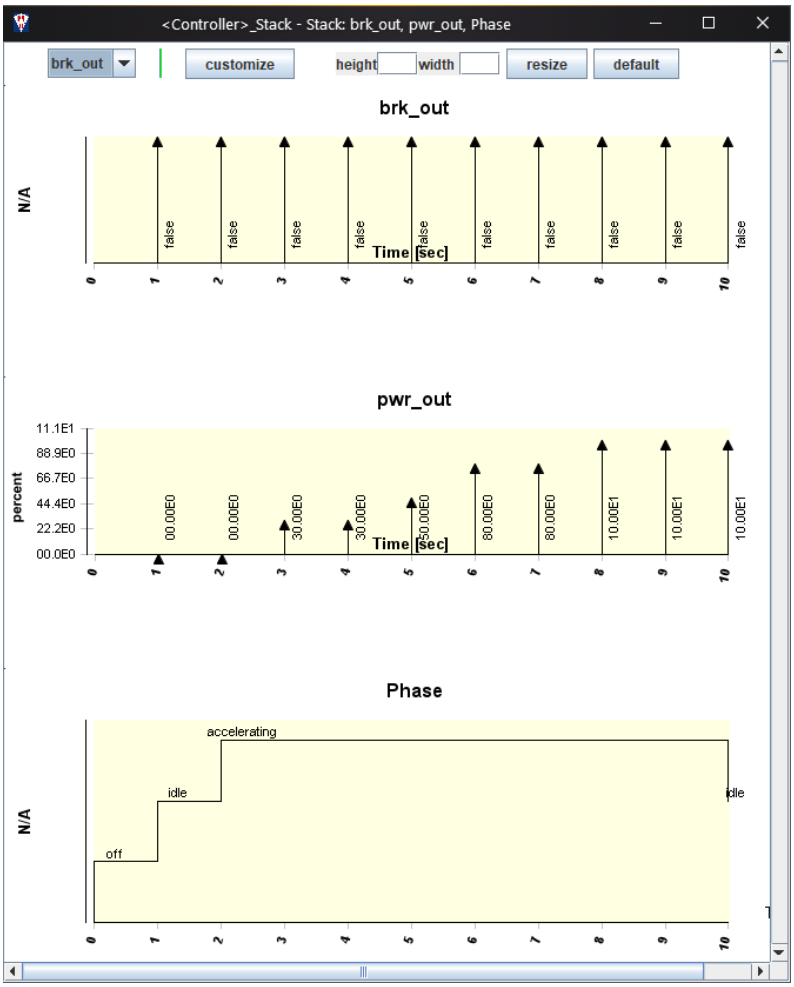

Figure 22. Part of the output trajectory and phase changes in the first experiment

Graphical integration of DEVS-FMI is not supported. The input received from FMU in the controller is used and plotted in Figure 23. The full results, including input received and output sent at each step, are shown in APPENDIX VI. Some observations can be made about the data.

- The interaction between DEVS controller model and FMU can be observed and verified within the DEVS-FMI. Every step, the controller sends outputs based on the inputs arriving from the user input module as

well as the FMU. On the next step, DEVS-FMI collects the outputs produced by the DEVS model and forward them into the FMU. The FMU then reports updated motor speed as well as other states to the DEVS-FMI, which is processed and injected into the DEVS model.

- The acceleration and deceleration are not effective instantaneously – for example, the controller received input to accelerate at 30 duty cycle in step 2, but the scooter's reaction is "delayed" until the next step. This is because the output produced by the controller is not processed until the beginning of the next step.
- Instead of using raw data for motor speed and battery charge (such as speed of the motor in rad/s and remaining battery in Ws), they are transformed into other representations to facilitate simulation. For battery, the number indicates the percentage of the battery remaining (e.g., 0.9906 means 99.06% battery remains), and the motor speed is converted into the speed of the scooter in mph. This conversion is done in the electric scooter co-simulation module. As the battery size and the wheel diameter can vary from one scooter to another, more abstract units mean different types of electric scooter can use the same DEVS models, with possibly fewer changes in the electric scooter co-simulation module.

7.4.2. Experiment 2: A Profile Showing Possible State Transitions

To verify that the DEVS speed controller model follows the behavior shown in Figure 11, a separate user input profile is designed and shown below. This input profile is meant to trigger all the states (except for PowerSaving) and examine if the state transition works as designed.

Second 1. User accelerates at MAX (100) rate.

Second 2. User accelerates at MAX (100) rate.

Second 3. User let the scooter coast (OFF).

Second 4. User brakes.

Second 5. User let the scooter coast (OFF).

Second 6. User brakes.

Second 7. User accelerates at MAX (100) rate.

Second 8. User breaks.

Second 9. User exits the scooter (idle)

The speed controller model was modified so that it only outputs brk_out when there is a change. The input trajectories are shown in Figure 23, and the output trajectories are shown in Figure 24. The following findings can be noted:

- The phases of the model closely follow the state machine diagram in Figure 11. This indicates that the DEVS formalism is able to capture the dynamics of the simplified electric scooter system.

- The inputs from the motor arrives at regular discrete-time steps with 1 time-unit intervals, despite the FMU takes irregular amount of time to execute. DEVS-FMI transforms the responses from the FMU into discrete time-interval messages that DEVS can work with.
- The brk_out output only occurs when its value is changed. Thus, the outputs of a DEVS model are not tied to regular intervals.

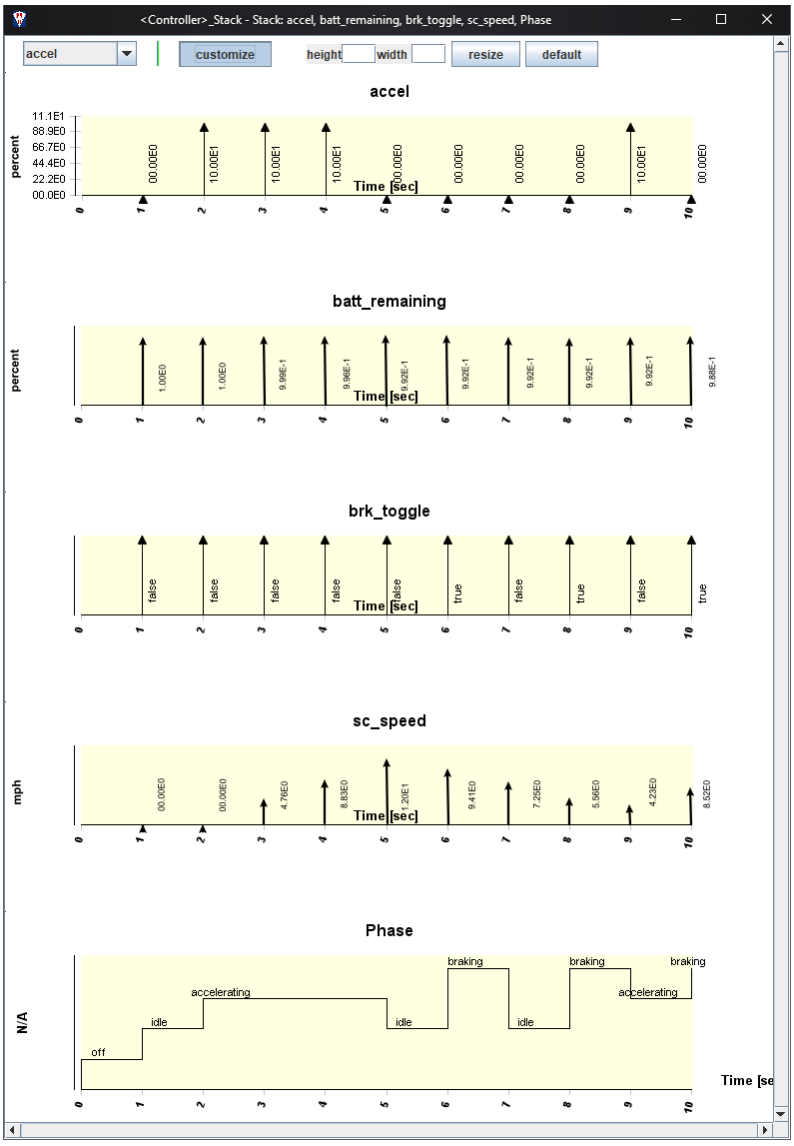

Figure 23. Trajectory of inputs and phase transitions in the second experiment

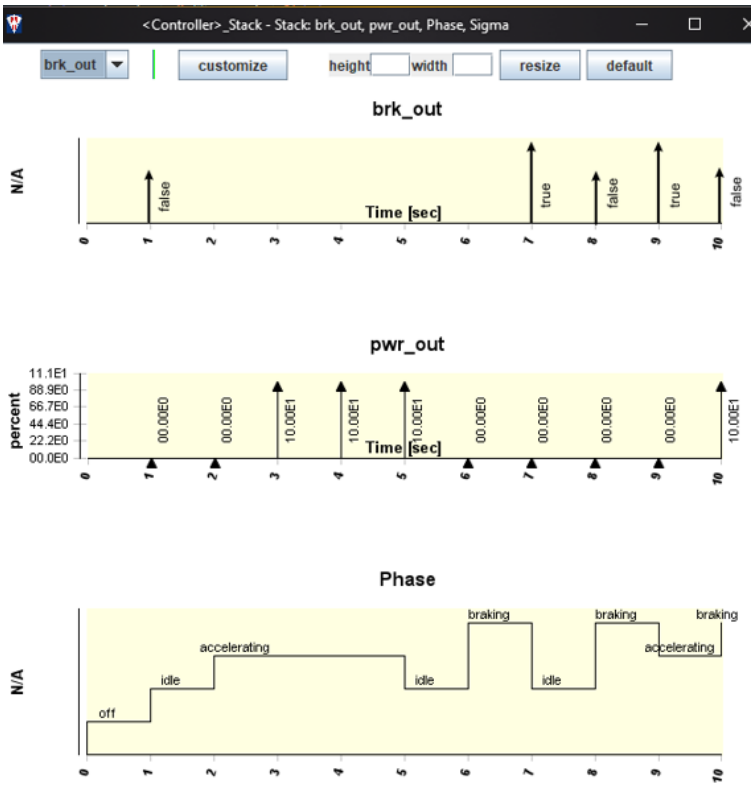

Figure 24. Trajectory of outputs and phase transitions in the second experiment

7.5. Impact of the DEVS-FMI Adapter (I/O Devices)

It is necessary to highlight the impact of the DEVS-FMI Adapter on the performance and accuracy of the overall simulation. Since the DEVS-FMI situates in between physical and cyber parts and transforms the messages between them, it can be considered as input and output devices. Therefore, the performance impact of the DEVS-FMI adapter also represents the impact of the I/O devices in the scooter co-simulation model. Likewise, the accuracy of the DEVS-FMI adapter represents the accuracy of the I/O devices in the co-simulation.

7.5.1. Performance

The efficiency of the DEVS-FMI adapter is crucial to its usability. To investigate the performance impact of the DEVS-FMI adapter, the following three simulation scenarios are devised and evaluated.

1. Normal co-simulation setup (identical to Experiment 1: A Typical User Input Profile).
2. Identical to (1) but uses a dummy FMU that is compatible with the electric scooter FMU. It simply returns constant values when queried for state values and accepts values from the DEVS simulator but does not act upon them. This way, the DEVS-FMI adapter will still be used but simulation time of the FMU will be kept to a minimum.
3. Similar to (1) but with DEVS-FMI integration disabled. Instead of having FMU providing the electric scooter speed and battery level information to the DEVS model, the UserInput model is altered to send dummy values for both. Therefore, this scenario involves only communications between DEVS and DEVS-FMI. This scenario is used as the baseline for the DEVS-FMI execution time as measured using the computer's clock.

Each of the three scenarios are simulated for 10 logical-time steps in DEVS-Suite, and the actual execution time consumed to simulate each step is measured. Each scenario is repeated five times, and average of these values is taken. In addition, the SimView and

tracking visualizations are turned off. The results are shown in Figure 25. (Full results are available in APPENDIX VII)

Figure 25. Execution time of the three simulation scenarios

According to the simulation results, the following observations can be made.

- For both scenarios with FMU enabled, the first step takes significantly longer than subsequent steps. This may be caused by initialization of the associated FMU on the first step. The DEVS-only model also consumes comparably more time in the first step.
- In scenario 1, the FMU (OpenModelica) uses the majority of overall simulation (execution) time. This suggests that the performance of the FMU has a greater impact on the overall performance.

- The DEVS-FMI adds little overhead to the computing time used for the co-simulation. The differences in execution time between the scenario with dummy FMU and no FMU is negligible.
- These experiments are conducted with 1 second logical-time FMU simulation length. Further testing is needed to investigate the performance of the co-simulation in shorter simulation lengths, such as 1 millisecond. Specifically, 1) if the overhead of DEVS-FMI becomes more prominent, and 2) how the FMU execution time changes in response to these changes. Chapter 7.6 provides experiments and preliminary observations regarding these questions.

7.5.2. Data Accuracy of Single and Double Precision on DEVS-FMI

Apart from the accuracy losses due to step size settings in FMU execution, the DEVS-FMI can also introduce some inevitable losses due to data conversion between C (FMU) and Java (DEVS-Suite). To investigate this further, an experiment is set up to capture the differences of the motor speed between OpenModelica and what received in the Java-FMI. The step size is fixed at 1.0E-05 seconds for all simulations, and simulations of various lengths are run. In consideration of other use scenarios, such as machines with limited memory, the simulation is run with results saved in double and float types.

Figure 26 shows the error values for five logical-time simulation durations. The error rate ranges from 0.862% to 1.048%, which is in line with findings in section 7.3. As

expected, the use of float data type incurs additional error when compared to the results using double type, which may be used as a substitute data type in simulators that have access to limited memory and processing power.

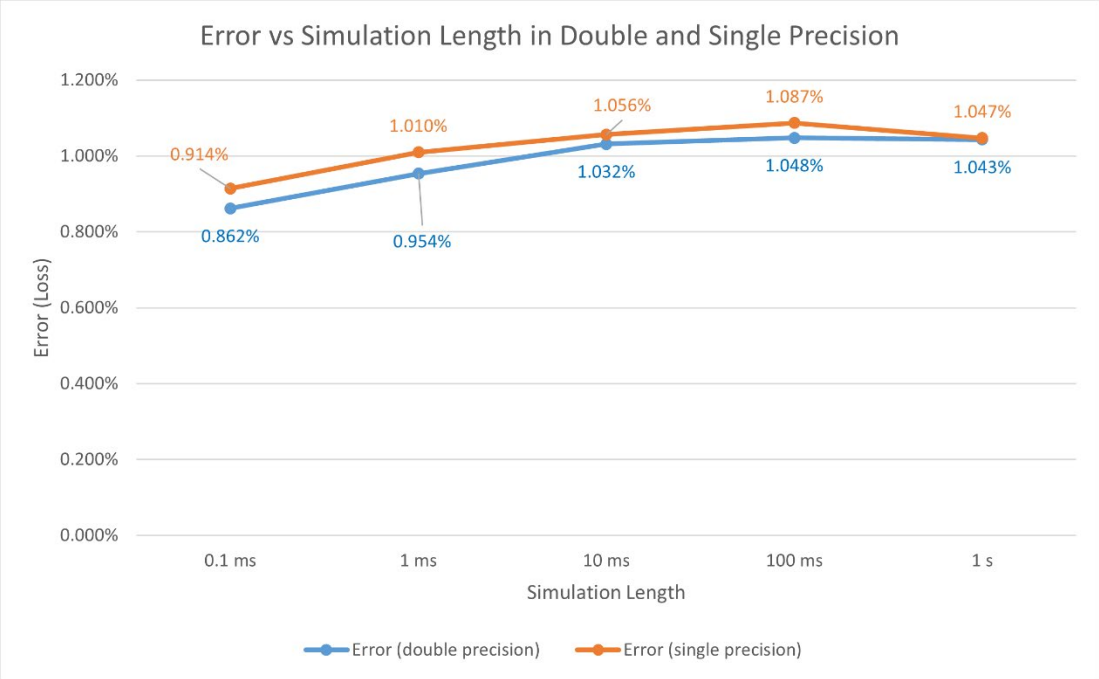

Figure 26. Error of different simulation lengths in DEVS-FMI

7.6. Co-simulation Performance

Apart from the performance of the DEVS-FMI, it is also important to test the execution performance (i.e., speedup factor) of this co-simulation approach if it were to be executed as-fast-as-possible. Because there are three parts in the DEVS-FMI co-simulation approach, it is necessary to discuss the performance of each constituent parts before analyzing them holistically.

7.6.1. DEVS-Suite (Cyber Part)

There is a built-in real-time factor slider on the DEVS-Suite interface (shown at lower-left corner of Figure 21) that controls the speedup factor of the DEVS logical time with regards to real time (i.e., JVM clock). This is achieved by introducing a pause in between each simulation steps. The slider value goes from 1.0E-04 (0.0001x sigma) to 1000. In essence, at smaller values, the speedup is constrained by the actual simulation time of the DEVS models, which is not affected by the slider configuration.

The third simulation setup described in section 7.5 (i.e., SimView and tracking are turned off) is used to assess the time used by the DEVS model in different real time factor settings. The model is executed with six different real-time factors: 1.0E-4 (DEVS-Suite default), 0.001, 0.01, 0.1, 0.5, and 1.0. For each real-time factor, the model would execute for five steps, and the average of these five steps is then compared.

As shown in Figure *27*, the total execution times (orange blocks) are controlled by the real-time factor settings. The delay due to the increase in each real-time factor value (grey blocks) increases the total simulation execution time. (Note that the data is scaled by 10 times and presented in log scale to aid with visualization).

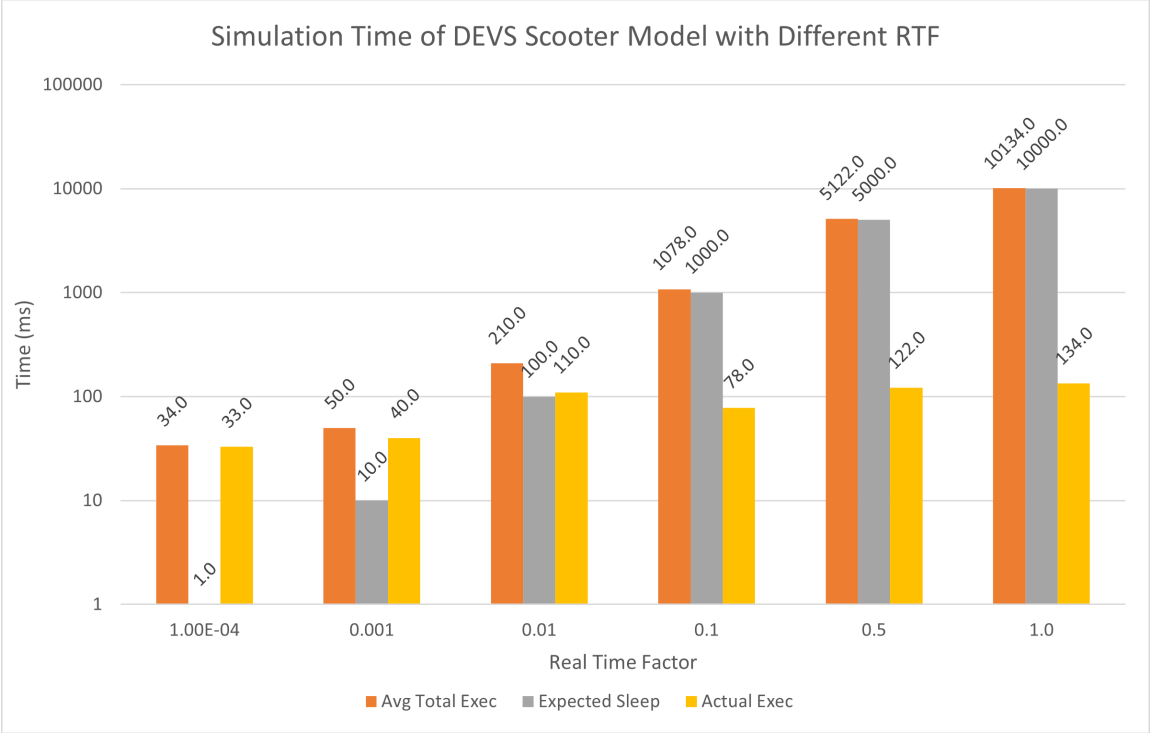

Figure 27. Simulation time of the DEVS scooter model with six different real time factors

7.6.2. FMUs (Physical Part)

It is possible to "speed-up" the FMU simulation by using longer step sizes (see section 7.3). However, the speed-up gain comes at proportionally higher loss of accuracy. Changing the length of each FMU simulation segment may change the overall co-simulation execution speed due to the overheads of the DEVS-FMI and the JavaFMI library.

To investigate the relationship of execution time and simulation length of the FMU with the DEVS-FMI, the electric scooter FMU is executed in seven different logical-time simulation lengths (durations): 1.00E-04, 0.001, 0.01, 0.1, 0.5, 1 (used in electric scooter study), and 5 seconds. As in the electric scooter study, the step size is set

to 1.0E-5 seconds for all seven experiments. Five simulation steps are run for each simulation length, and the average time consumed for these steps is calculated. Note that the time for the first segments is not included in the average, because the initialization of the FMU adds a significant overhead to the simulation that does not show up in subsequent steps.

Figure 28 shows the result for each of these simulations. As expected, the execution time increases mostly linearly with the simulation length, as evidenced by the results for 0.1 – 5 second settings.

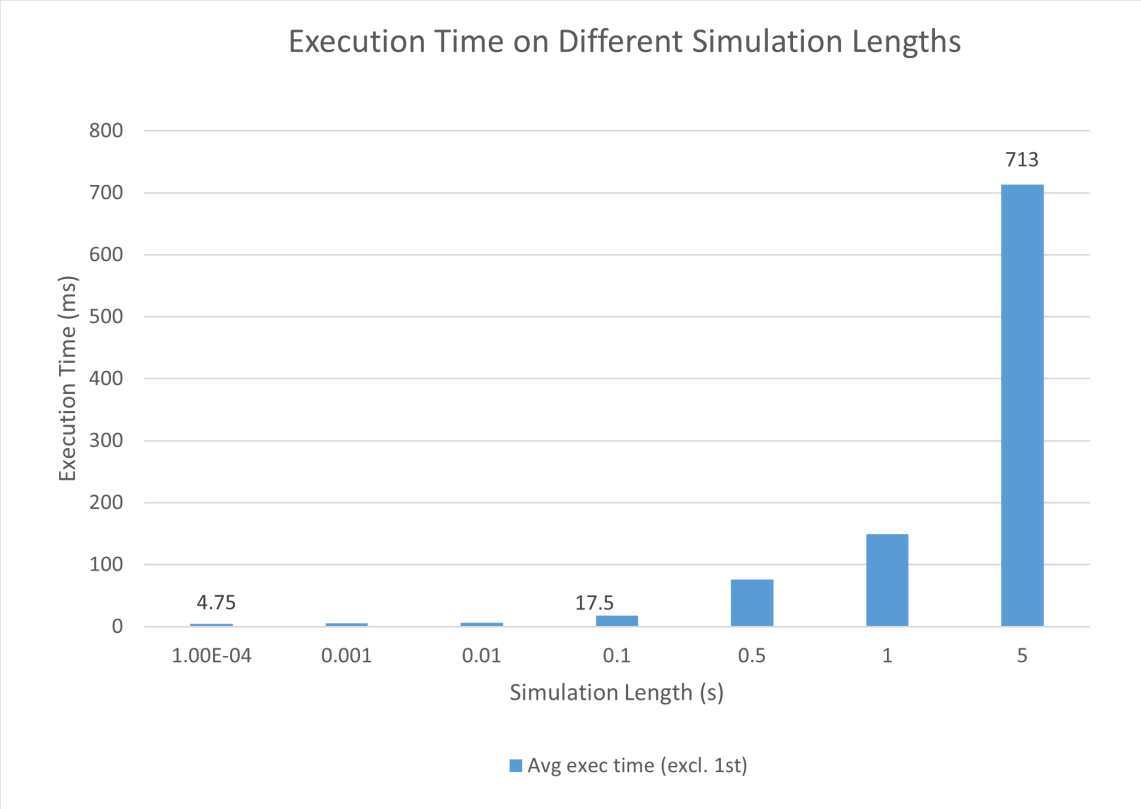

Figure 28. Average execution time on different simulation lengths

7.6.3. Observation

It is clear that the following configurations can help speed up the logical simulation.

- Use lower real time factor in DEVS-Suite (however, at lower settings the time consumption is bounded by the actual Java implementation).
- Use longer simulation lengths for FMU (at the cost of less precise responses from FMU).

Clearly, the following configuration can cause logical simulations to take longer than actual time to execute.

- Use higher real-time factor in DEVS-Suite.
- Use shorter simulation lengths for FMU (improves the precision of the responses).

8. Conclusion and Future Work

This project has demonstrated co-simulation for Parallel DEVS models and FMUs with the benefit of the Four-Variable model. By applying the Four-Variable model, the FMI 2.0 standard, and a rigorous round-robin synchronization scheme critical to correct interaction, the DEVS-FMI 2.0 is developed for the DEVS-Suite simulator and used with the OpenModelica simulator.

The DEVS-FMI augments the existing DEVS-Suite simulator with the improved FMI 2.0 standard. The DEVS-Suite serves as the master simulator which can control the operation flow for slave FMUs. The logic for the DEVS to interact with FMUs is specified in the DEVS-FMI and a corresponding co-simulation module.

DEVS-FMI enables two-way round-robin communication of simulation execution cycles between DEVS models and individual FMUs. The JavaFMI library is used to obtain information from and sending information to each FMU, and the DEVS-FMI provides an abstraction on top of JavaFMI to facilitate input/output message communication to DEVS models. Synchronization is supported through a “stop-and-go” routine for an FMU, as the FMU must complete its execution before it can respond to new inputs. In principle, before each step in a DEVS model is executed, its associated FMUs are executed for a certain period. Some states are saved from the FMUs and used to initialize the next FMU execution, and some states are used as output from the FMU. The output is processed and sent to the DEVS model, and after the step is executed, the

output from the DEVS model is processed and sent to the FMUs to initialize the next round of execution, along with the saved states.

A portion of an electric scooter representing its cyber and physical components is selected, modeled, simulated, and evaluated. This model demonstrates the feasibility and effectiveness of the DEVS-FMI adapter interface for use with OpenModelica. It shows the DEVS-FMI coordinates the discrete-time simulations of the electric scooter's parallel DEVS and Modelica models, developed in the DEVS-Suite and OpenModelica tools, with accuracy (depending, in part, on the FMU step size setting).

Currently, there is no GUI support for the DEVS-FMI interface in the DEVS-Suite simulator. This can complicate the setup of co-simulation, and visualizing any FMU's behavior, except its inputs and outputs accessed in DEVS-FMI and declared as DEVS inputs and outputs. To solve this usability limitation, research is needed on extending the DEVS-Suite simulator.

The operation of the DEVS-FMI assumes that implicit state-events in the FMUs occur on specified time intervals. This assumption would work for some scenarios, including the electric scooter example, because the state-events for the coupled DEVS model and the FMU for the motor and battery are defined to occur at the same, regular time intervals. For realistic cases, some approximation algorithm is needed to localize the event times for the FMU relative to the event times for the DEVS model.

The DEVS-FMI is not designed to support co-simulation for multiple FMUs. Each FMU must be tied to a DEVS model for co-simulation (i.e., one-to-one

relationship). Some other critical features, such as FMU dependency solving and inter-FMU message exchange, also needs to be incorporated to support multi-FMU simulation.

From a broad perspective, the Architecture and Analysis Design Language (AADL) is a text-based modeling language that is particularly tailored towards time and safety-critical systems [32]. It is standardized by the Society of Automotive Engineers (SAE) and has seen widespread uses in aerospace and automotive scenes. It has a simple visual representation for the models, which is based on the text specifications. It supports a model-based development approach throughout the system lifecycle [33]. An AADL DEVS Annex [34] is proposed for behavior modeling. The AADL-DEVS language supports hierarchical, modular discrete event modeling and simulation. The Open Source AADL Tool Environment (OSATE) [35] is extended to support model code generation for the DEVS-Suite simulator. The OSATE and the DEVS-Suite supported with DEVS-FMI can be useful toolchain for analyzing and designing CPS and can be a topic of interest for further research.

References

[1] A. T. Al-Hammouri, M. S. Branicky and V. Liberatore, "Co-Simulation Tools for Networked Control Systems," in *International Conference on Hybrid Systems: Computation and Control (HSCC 08)*, St. Louis, 2008.

[2] T. Blockwitz, M. Otter, J. Akesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson and A. Viel, "Models, Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation," in *9th International Modelica Conference*, München, 2012.

[3] P. Fritzson, P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli and D. Broman, "The OpenModelica Modeling, Simulation, and Development Environment," in *46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005)*, Trondheim, 2005.

[4] B. P. Zeigler, H. Praehofer and T. G. Kim, Theory of Modeling and Simulation, London: Academic Press, 2000.

[5] S. Kim, H. S. Sarjoughian and V. Elamvazhuthi, "DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring," in *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, 2009.

[6] E. Widl and W. Müller, "Generic FMI-compliant Simulation Tool Coupling," in *Proceedings of the 12th International Modelica Conference*, Prague, 2017.

[7] J. J. Hernández-Cabrera, J. Évora-Gómez and O. Roncal-Andrés, "javaFMI," SIANI, University of Las Palmas, 11 August 2020. [Online]. Available: https://bitbucket.org/siani/javafmi/wiki/Home. [Accessed 23 October 2020].

[8] S. P. Miller and A. C. Tribble, "Extending the four-variable model to bridge the system-software gap," in *20th DASC. 20th Digital Avionics Systems Conference*, Daytona Beach, 2001.

[9] M. H. Quraishi, H. S. Sarjoughian and S. Gholami, "Co-Simulation of Hardware RTL and Software System Using FMI," in *2018 Winter Simulation Conference (WSC)*, Gothenburg, 2018.

[10] E. Kofman and S. Junco, "Quantized-state systems: a DEVS Approach for continuous system simulation," *Transactions of the Society for Computer Simulation International,* vol. 18, no. 3, pp. 123-321, 2001.

[11] A. C. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism," in *Proceedings of Winter Simulation Conference*, Lake Buena Vista, 1994.

[12] H. S. Sarjoughian, "Restraining complexity and scale traits for component-based simulation models," in *2017 Winter Simulation Conference (WSC)*, Las Vegas, 2017.

[13] ACIMS, "DEVS-Suite," Arizona State University, September 2020. [Online]. Available: https://acims.asu.edu/software/devs-suite/. [Accessed 15 January 2020].

[14] A. Junghanns and T. Blochwitz, "10 Years of FMI: Where are we now? Where do we go?," FMI Modelica Association Project, 2018.

[15] P. Bunus and P. Fritzson, "Automated Static Analysis of Equation-Based Components," *SIMULATION: Transactions of The Society for Modeling and Simulation International,* vol. 80, no. 7-8, pp. 321-345, 2004.

[16] R. Alur, Principles of Cyber-Physical Systems, Cambridge, MA: The MIT Press, 2015.

[17] H. S. Sarjoughian, C. Zhang and X. Lin, *Control and Decision Communication Across Heterogeneous Model Types,* Linköping: Model-Based Cyber-Physical Product Development Workshop, 2021.

[18] H. Szczerbicka, K. S. Trivedi and P. K. Choudhary, "Discrete Event Simulation with Application to Computer Communication Systems Performance," in *Information Technology*, Boston, 2004.

[19] B. Wang and J. S. Baras, "HybridSim: A Modeling and Co-simulation Toolchain for Cyber-physical Systems," in *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, Deft, 2013.

[20] D. L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering, Vol. 2," McMaster University, Hamilton, 1991.

[21] N. Ulfat-Bunyadi, R. Meis and M. Heisel, "The Six-Variable Model - Context Modelling Enabling Systematic Reuse of Control Software," in *Proceedings of the 11th International Joint Conference on Software Technologies - Volume 2*, Lisbon, 2016.

[22] E. Widl, W. Müller, A. Elsheikh, M. Hörtenhuber and P. Palensky, "The FMI++ library: A high-level utility package for FMI for model exchange," in *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, Berkeley, 2013.

[23] H. S. Sarjoughian, Y. Chen and K. Burger, "A component-based visual simulator for MIPS32 processors," in *2008 38th Annual Frontiers in Education Conference*, Saratoga Springs, 2008.

[24] B. Camus, V. Galtier and M. Caujolle, "Hybrid Co-simulation of FMUs using DEV&DESS in MECSYCO," in *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, Pasadena, 2016.

[25] T. S. Nouidui, J. Coignard, C. Gehbauer, M. Wetter, J.-Y. Joo and E. Vrettos, "CyDER – an FMI-based co-simulation platform for distributed energy resources," *Journal of Building Performance Simulation,* vol. 12, no. 5, pp. 566-579, 2019.

[26] V. Galtier, S. Vialle, C. Dad, J.-P. Tavella, J.-P. Lam-Yee-Mui and G. Plessis, "FMI-based distributed multi-simulation with DACCOSIM," in *DEVS '15: Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, Alexandria, 2015.

[27] Dassault Systèmes, "FMPy," 27 10 2020. [Online]. Available: https://github.com/CATIA-Systems/FMPy. [Accessed 2 11 2020].

[28] Modelon AB, "FMI Library: part of JModelica.org," 6 11 2017. [Online]. Available: https://jmodelica.org/fmil/FMILibrary-2.0.3-htmldoc/index.html. [Accessed 2 11 2020].

[29] L. I. Hatledal, H. Zhang and A. Styve, "FMI4j: A Software Package for working with Functional Mock-up Units on the Java Virtual Machine," in *Proceedings of The 59th Conference on Simulation and Modelling (SIMS 59)*, Oslo, 2018.

[30] J. Hughes, "fmi - crates.io," 28 October 2020. [Online]. Available: https://crates.io/crates/fmi. [Accessed 19 November 2020].

[31] Modelon AB, "Overview - PyFMI 2.5 Documentation," Modelon AB, [Online]. Available: https://jmodelica.org/pyfmi/. [Accessed 19 November 2020].

[32] SAE International, "AS5506C: Architecture Analysis & Design Language (AADL) - SAE International," 8 January 2017. [Online]. Available: https://www.sae.org/standards/content/as5506c/. [Accessed 8 March 2020].

[33] J. Delange, AADL In Practice, Reblochon Development Co, 2017.

[34] E. R. Ahmad, B. C. Larson, S. Barrett, N. Zhan and Y. Dong, "A Behavior Annex For AADL Using The DEVS Formalism," *2019 Spring Simulation Conference (SpringSim),* 10 June 2019.

[35] Carnegie Mellon University, "Welcome to OSATE," 30 October 2020. [Online]. Available: https://osate.org/. [Accessed 24 November 2020].

APPENDIX I

BASIC ATTRIBUTES OF AN ELECTRIC SCOOTER

The following paragraphs describe some basic parameters and attributes of a typical electric scooter that is used for study and demonstration purposes. The construct of these parameters attempts to follow the CPS-specific Four-Variable model established in the introductory sections, while remaining true to the actual specification of a scooter as much as possible.

Components Considered

For brevity, only the following components are included in the modeling & simulation process to demonstrate the DEVS-FMI extension. They are electric motor, battery, brake handle, acceleration handle, and software controller. Other components, such as wheels and frame of the scooter, are considered less relevant to the study and will not be modeled in detail (or not modeled at all).

Environment

Table 1 shows some important variables in the scooter system that is being observed. These variables are fundamental to the system and are influenced by output from actuators.

Table 1. Select environment variables for the scooter

Name	Range of values	Unit	Type
motorSpeed	{x | 0 <= x <= 2600}	rpm	real
remainingCapacity	{x | 0 <= x <= 777600}	Ws	real

motorSpeed indicates the speed at which the motor revolves, measured in rpm (revolutions per minute). Since the motor is attached to a wheel, speed of the motor directly affects speed at which the wheel turns, which in turn affects speed of the scooter.

In this study, the radius of the wheels is assumed to be 1.5 inches, and that only input voltage is considered for determining the motor speed. Other factors, such as friction and inertia, are not considered. Specifically, motor would consume the electricity provided by battery and convert it into mechanical energy (i.e., rotation of the shaft). This rotation propels the wheel to rotate forward, causing the scooter to accelerate. Even though the relationship between the motor speed and the voltage is approximately linear, because of inertia of the scooter and the motor itself, the motor speed does not necessarily follow a linear relation. Therefore, it is necessary to account for the inertia in the model to yield more accurate results.

remainingCapacity represents the amount of electricity available in the battery, expressed in Ws (watt-second). This quantity is derived from the actual capacity of the battery (9,000 mAh or 9 Ah) multiplied by nominal voltage of the battery (24 V), which becomes 216 Wh. Multiply that quantity by 3,600 to get the final result of 777,600 Ws. The measurement in Ws is easier to work with as Modelica models simulates in real time, and battery capacity changes (derivatives) can be simply expressed in the product of current and voltage. An integral over the wattage yields watt-hour energy that battery has used, which can be used to determine the remaining capacity. As electricity is consumed by the motor, the available capacity of the battery decreases over time.

Note that battery acts as both an actuator and part of the environment. This duality stems from the way the battery is constructed – battery can act on motor to drive it forward, while its remaining capacity is also affected by how much electricity it used.

Sensors (Input Devices).

Table 2 and Table 3 show how analog data obtained from the environment is translated into corresponding digital signal for further processing. There are two types of sensors, namely speed sensor and battery sensor. Note that user-initiated inputs are listed separately as they are not strictly part of the Four-Variable model.

Table 2. Input and output parameters of the speed sensor

Name	Range of values	Unit	Type	Direction
motorSpeed	{x | 0 <= x <= 2600}	rpm	int	in
scooterSpeed	{x | 0 <= x <= 25}	mph	real	out

Table 3. Input and output parameters of the battery sensors

Name	Range of values	Unit	Type	Direction
remainingCapacity	{x | 0 <= x <= 777600}	Ws	real	in
batteryRemaining	{x | 0 <= x <= 100}	percent	real	out

The speed sensor monitors and reports the speed of the scooter to the controller. Even though movement of the wheels are continuous, the speed sensor reports the speed at a fixed rate (as it is impractical to make it continuous). scooterSpeed indicates measured speed of the scooter, relative to the ground, in any given moment. The speed is measured in mph (miles per hour). In a realistic setting, the measured speed can be affected by a number of factors, such as the speed of the motor, whether the brake is engaged, rider weight (scooter load), ground terrain, friction, and many more. For the purpose of the study, only three factors are considered: speed of the motor, brake engagement, and scooter load (static constant). Other factors are ignored (e.g., energy lost due to friction) or considered “perfect” (e.g., flat terrain without obstacle).

The battery sensor is an embedded sensor in the battery that monitors and reports the remaining battery capacity. As the nominal battery capacity is 777,600 Ws, the remaining percentage (batteryRemaining) can be simply the remaining capacity divided by the nominal capacity.

Another class of input variable is initiated by the user and may arrive spontaneously. These are listed in Table 4.

Table 4. User-initiated input variables

Name	Range of values	Unit	Type	Direction
acceleration	{x | 0 <= x <= 100}	percent	int	out
brakeToggle	{true, false}	N/A	bool	out

The acceleration handle is a device that controls the acceleration of the scooter. It can be a handle that, when depressed, makes the scooter accelerate. The more user depresses the handle, the faster the scooter should accelerate.

The brake handle is a handle that is used to control the brake. When the brake handle is depressed, the scooter will attempt to slow down by engaging the brake and cut off power to the motor. The braking intensity would be constant regardless of the handle depression.

Software

The software controller receives and controls the speed of the scooter. When a user depresses the accelerator, the controller will attempt to increase the speed of the scooter by increasing power output. When a user depresses the brake handle, it will engage the brake to slow down the scooter. The speed of the scooter will not exceed a preset safety limit (such as 15mph). When this limit is exceeded, the controller will

attempt to slow down the scooter by cutting off the power to motor and, when necessary, engaging the brake. Table 5 shows relevant variables in the software system.

Table 5. Variables in the software controller

Name	Range of values	Unit	Type
scooterSpeed	{x | 0 <= x <= 25}	mph	real
batteryRemaining	{x | 0 <= x <= 100}	percent	real
brakeControl	{true, false}	N/A	bool
dutyCycle	{x | 0 <= x <= 100}	percent	real

Note the notion of dutyCycle. The software controls the speed of motor by changing the voltage output to the motor. It is usually done with a technique called pulse-width modulation (PWM), where signal (output) can only be high (i.e., 24V) or low (i.e., ground) at any time. However, the proportion of time the signal is high relative to low can be changed, thereby changing the effective voltage that motor receives. For example, to achieve a desired output of 12 volts, the signal can be set to 50% duty cycle so that the signal is on high 50% of time in a given period. Therefore, a correspondence of dutyCycle percentage and effective voltage that motor receives can be established.

Actuators (output devices)

Finally, actuators translate the control signals they receive from the software and translate them into some changes in the environment. Table 6 and Table 7 show the relevant input and output parameters of the battery and the brake.

Table 6. Input and output parameters of the battery

Name	Range of values	Unit	Type	Direction
dutyCycle	{x | 0 <= x <= 100}	percent	real	in
voltageOutput	{x | 0 <= x <= 24}	volt	real	out

The battery stores electrical energy that can be used to power the motor. The higher the voltage output, the more torque the motor generates. The motor, in turn, accelerates the scooter. The voltage is regulated by the software with PWM. To motor, this should be transparent as it "perceives" the width-modulated signals the same way as analog signals.

Table 7. Input and output parameters of the brake

Name	Range of values	Unit	Type	Direction
brakeControl	{true, false}	N/A	bool	in
isBraking	{true, false}	N/A	bool	out

The brake can be any physical device that slows and stops the scooter. Usually, the brake is attached on the wheel and applies friction to one of the wheels when activated. The friction produced is used to slow down the wheel hence the scooter.

APPENDIX II

SETUP OF THE ELECTRIC SCOOTER MODEL IN OPENMODELICA

The following tables list some important parameters that is used to set up the electric scooter model in Modelica, as discussed in Modelica Electric Scooter Model and Figure 12. These parameters attempt to recreate the real-life electric scooter to the extent possible.

Table 8 shows some parameters for dcpmData that is used to control some behavior of the electric motor.

Table 8. Select parameters of dcpmData

Name	Value	Unit	Description
Jr	0.1	$kg.m^2$	Rotor's moment of inertia
VaNominal	24	V	Nominal armature voltage
IaNominal	13.5	A	Nominal armature current
wNominal	2600	rev/min	Nominal speed of the motor

Table 9 shows some parameters of the pulse signal. This is used as input to generate PWM-modulated voltage input to the motor. Note the period is set to 2 milliseconds – this corresponds to setup in similar electric scooters.

Table 9. Select parameters for pulse signal

Name	Value	Unit	Description
amplitude	24	V	Amplitude of the pulse (effectively voltage on high)
width	dutyCycle	percent	The width of pulse in % of period (effectively duty cycle, set by external parameter dutyCycle)
period	0.002	s	Time for one period

Battery related configurations are shown in Table 10. The initial value y is set to parameter prevBattLevel. This indicates that it is set by this parameter each time the simulation starts, similar to dutyCycle.

Table 10. Select parameters in batteryLevel

Name	Value	Unit	Description
k	-1	N/A	Integrator gain (i.e., coefficient of the integration)
y_start	prevBattLevel	Ws	Initial y value (effectively initial battery level)

Parameters of some other components are shown below in Table 11. Initial phi value is determined by phi.start, which is passed in as parameter liPhi when simulation starts.

Table 11. Select parameters of loadInertia and loadTorqueSetup

Name	Value	Unit	Description	Component
J	0.5	$kg.m^2$	Moment of inertia	loadInertia
phi.start	liPhi	rad	Initial rotation angle of component	loadInertia
stepTorque	-0.86	N.m	Height of torque step	loadTorqueSetup

APPENDIX III

SIMULATION PLATFORM

The following listings show the hardware and software environment used in experiments.

Hardware

- CPU: AMD Ryzen 9 3900X 12-Core @ 3.79 GHz
- Memory: 64 GB DDR4 3600 MHz

Software

- Operating System
  - Windows 10 64-bit (Build 19041.572)
- DEVS simulator
  - Java 11.0.8 64-bit
  - DEVS-Suite 6.0.0
  - fmu-wrapper 2.26.3 (part of JavaFMI)
- FMU
  - OpenModelica v1.14.1 64-bit
  - OMSimulator v2.1.0-dev-147

APPENDIX IV

RESULT SET 1 – COMPARISON OF SIMULATION ACCURACY ON THREE SIMULATION SETUPS

Table 12 shows the results of simulating an identical electric scooter model in three different setups, as prescribed in Continuity and Accuracy of FMU Simulations section. Diff indicates the difference between the output of both FMUs and the "reference" OpenModelica configuration.

Table 12. Results of dcpm.wMechanical on three simulation setups

Time (s)	OpenModelica	JavaFMI (normal)	JavaFMI (stop & restart)	Diff
0.0	0.00000	0.00000	0.00000	0.00%
0.2	5.47289	5.53024	5.53024	1.05%
0.4	11.6637	11.7856	11.7856	1.05%
0.6	17.5580	17.7416	17.7416	1.05%
0.8	23.1688	23.4108	23.4108	1.04%
1.0	28.5098	28.8075	28.8075	1.04%
1.2	33.5939	33.9447	33.9447	1.04%
1.4	38.4335	38.8348	38.8348	1.04%
1.6	43.0403	43.4897	43.4897	1.04%
1.8	47.4255	47.9207	47.9207	1.04%
2.0	51.5998	52.1386	52.1386	1.04%
2.2	55.5734	56.1536	56.1536	1.04%
2.4	59.3558	59.9755	59.9755	1.04%
2.6	62.9563	63.6135	63.6135	1.04%
2.8	66.3836	67.0767	67.0767	1.04%
3.0	69.6461	70.3722	70.3722	1.04%
3.2	72.7517	73.5112	73.5112	1.04%
3.4	75.7079	76.4983	76.4983	1.04%
3.6	78.5220	79.3417	79.3417	1.04%
3.8	81.2007	82.0483	82.0483	1.04%
4.0	83.7505	84.6248	84.6248	1.04%
4.2	86.1777	87.0772	87.0772	1.04%
4.4	88.4882	89.4119	89.4119	1.04%
4.6	90.6876	91.6339	91.6339	1.04%
4.8	92.7811	93.7494	93.7494	1.04%
5.0	94.7740	95.7629	95.7629	1.04%

APPENDIX V

RESULT SET 2 – COMPARISON OF SIMULATION ACCURACY ON JAVAFMI WITH SIX DIFFERENT STEP SIZES

Table 13 shows results from simulating an identical FMU in JavaFMI with different step sizes, as well as the corresponding execution time of each setups.

Table 13. Accuracy of simulation vs execution time with different step sizes

Time (s)	Open Modelica	1.00E-06	5.00E-06	1.00E-05	5.00E-05	1.00E-04	5.00E-04
0.0	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
0.2	5.47289	5.47860	5.50159	5.53024	5.75970	6.04710	8.33840
0.4	11.6637	11.6759	11.7246	11.7856	12.2735	12.8832	17.7599
0.6	17.5580	17.5763	17.6497	17.7416	18.4754	19.3927	26.7275
0.8	23.1688	23.1930	23.2899	23.4108	24.3790	25.2891	35.2654
1.0	28.5098	28.5396	28.6587	28.8075	29.9987	31.4874	43.3937
1.2	33.5939	33.6290	33.7693	33.9447	35.3485	37.1021	51.1309
1.4	38.4335	38.4736	38.6341	38.8348	40.4406	42.4467	58.4960
1.6	43.0403	43.0852	43.2650	43.4897	45.2877	47.5343	65.5069
1.8	47.4255	47.4750	47.6731	47.9207	49.9017	52.3771	72.1805
2.0	51.5998	51.6537	51.8692	52.1386	54.2938	56.9870	78.5331
2.2	55.5734	55.6314	55.8635	56.1536	58.4747	61.3767	84.5802
2.4	59.3558	59.4177	59.6656	59.9755	62.4544	65.5539	90.3364
2.6	62.9563	63.0220	63.2849	63.6135	66.2428	69.5301	95.8270
2.8	66.3836	66.4529	66.7301	67.0767	69.8490	73.3150	101.0430
3.0	69.6461	69.7188	70.0096	70.3722	73.2807	76.9179	106.0075
3.2	72.7517	72.8276	73.1314	73.5112	76.5484	80.3475	110.7330
3.4	75.7079	75.7870	76.1030	76.4983	79.6589	83.6121	115.2312
3.6	78.5220	78.6039	78.9317	79.3417	82.6197	86.7197	119.5132
3.8	81.2007	81.2854	81.6244	82.0483	85.4382	89.6779	123.5892
4.0	83.7505	83.8379	84.1876	84.6248	88.1211	92.4937	127.4691
4.2	86.1777	86.2677	86.6274	87.0772	90.6750	95.1742	131.1624
4.4	88.4882	88.5805	88.9500	89.4119	93.1060	97.7257	134.6870
4.6	90.6876	90.7822	91.1608	91.6339	95.4202	100.1545	138.0245
4.8	92.7811	92.8779	93.2653	93.7494	97.6230	102.4665	141.2101
5.0	94.7740	94.8729	95.2685	95.7629	99.7193	104.6662	144.2180
Avg Loss	0.00%	0.10%	0.52%	1.04%	5.22%	10.39%	52.21%
Exec Time (ms)	N/A	59754	12564	6670	1821	1193	697

APPENDIX VI

RESULT SET 3 – INPUTS, OUTPUTS, AND PHASE CHANGES OF THE DEVS ELECTRIC SCOOTER MODEL ON A GIVEN INPUT PROFILE

Table 14 shows various input and output of interest from the controller model. It also includes the current step (clock) and phase of the controller. Definitions of these input and output variables are explained in DEVS Electric Scooter Model section.

Table 14. Phase change, inputs, and outputs of controller model over time

		Inputs				Outputs	
Clock	Controller Phase	accel	batt_remaining	brk_toggle	sc_speed	brk_out	pwr_out
0	off	N/A	N/A	N/A	N/A	N/A	N/A
1	idle	0	0.9999	FALSE	0	N/A	N/A
2	accelerating	30	0.9999	FALSE	0	FALSE	30
3	accelerating	50	0.9997	FALSE	1.2359	FALSE	50
4	accelerating	80	0.9994	FALSE	2.4476	FALSE	80
5	accelerating	80	0.9986	FALSE	4.4099	FALSE	80
6	accelerating	100	0.9967	FALSE	7.4935	FALSE	100
7	accelerating	100	0.9943	FALSE	9.9460	FALSE	100
8	accelerating	100	0.9906	FALSE	12.852	FALSE	100
9	idle	0	0.9870	FALSE	15.151	FALSE	0
10	braking	0	0.9837	TRUE	16.948	TRUE	0
11	braking	0	0.9837	TRUE	13.277	TRUE	0
12	braking	0	0.9835	TRUE	10.268	TRUE	0
13	idle	0	0.9834	FALSE	9.1035	FALSE	0

APPENDIX VII

RESULT SET 4 – EXECUTION TIME OF THREE SIMULATION SETUPS

Table 15 shows the execution times measured using the JVM clock for three simulation setups to simulate 1 second in logical time, including the times used for initialization, execution of the associated FMU (if there is one), and execution of the DEVS model. Each simulation set-up is executed for 10 steps with 5 repetitions, resulting in 50 measurements for each setup. All measurements are in milliseconds.

Table 15. Execution time of various simulation setups

Actual FMU						
Step\Rep	1	2	3	4	5	Avg
1	703	690	692	690	700	695
2	174	170	171	166	173	170.8
3	179	165	164	165	174	169.4
4	167	165	171	165	172	168
5	173	162	165	164	173	167.4
6	168	160	165	163	170	165.2
7	170	163	165	163	167	165.6
8	172	164	164	163	170	166.6
9	170	161	165	160	168	164.8
10	170	163	168	164	167	166.4
					Avg (excl. step 1)	167.13
					Min (excl. step 1)	164.8
					Max (excl. step 1)	170.8
Dummy FMU						
Step\Rep	1	2	3	4	5	Avg
1	499	500	466	479	494	487.6
2	11	9	13	13	15	12.2
3	11	10	9	11	12	10.6
4	8	10	10	9	11	9.6
5	8	8	9	9	9	8.6
6	14	7	7	7	9	8.8
7	9	14	8	6	14	10.2
8	8	7	9	7	8	7.8
9	6	7	6	8	6	6.6
10	6	6	4	5	6	5.4

					Avg (excl. step 1)	8.867
					Min (excl. step 1)	5.4
					Max (excl. step 1)	12.2
No FMU						
Step\Rep	1	2	3	4	5	Avg
1	11	15	12	11	13	12.4
2	5	8	4	5	6	5.6
3	3	5	6	5	5	4.8
4	4	4	4	4	4	4
5	3	3	5	4	3	3.6
6	4	2	4	2	4	3.2
7	3	2	4	3	4	3.2
8	2	5	6	3	3	3.8
9	4	3	2	4	3	3.2
10	2	4	7	4	4	4.2
					Avg (excl. step 1)	3.956
					Min (excl. step 1)	3.2
					Max (excl. step 1)	5.6