LS-SA: Developing an FMI layered standard for holistic & efficient sensitivity analysis of FMUs

Tobias Thummerer[1] Hans Olsson[2] Chen Song[3] Julia Gundermann[4] Torsten Blochwitz[4] Lars Mikelsons[1]

[1]Chair of Mechatronics, University of Augsburg, Germany, {tobias.thummerer,lars.mikelsons}@uni-a.de
[2]Dassault Systemes AB, Sweden, {hans.olsson}@3ds.com
[3]ABB Corporate Research Center, Germany, {chen.song}@de.abb.com
[4]Keysight Technologies Deutschland GmbH, Germany,
{julia.gundermann,torsten.blochwitz}@keysight.com

Abstract

The Functional Mock-up Interface (FMI) is *the* standard for exchanging industrial simulation models in a variety of different applications. Although sensitivity analysis for continuously differentiable systems is directly supported by the standard, for systems with state discontinuities, it is only possible to determine correct sensitivities to a limited extent. In this position paper, we investigate how sensitivity analysis for discontinuous Functional Mock-up Units (FMUs), i.e. including state and time events, works in theory and which additional steps are required to obtain correct results in practice. We further investigate that these steps are unnecessarily computationally intensive from a mathematical point of view, but cannot be implemented in a more efficient way under the current restrictions of the standard. We therefore make a concrete proposal for the new *layered standard sensitivity analysis* (LS-SA) that remedies the current deficits of FMI in the sensitivity analysis of discontinuous systems. In this way, *LS-SA* opens FMI towards a variety of next-level applications — including (scientific) machine learning and optimal control — by providing fully differentiable FMUs under high computational performance.

Keywords: Functional Mock-up Unit, Functional Mock-up Interface, Discontinuous, Events, Machine Learning, Scientific Machine Learning, Neural FMU, Layered Standard

1 Introduction

We start by motivating the actual topic, followed by a brief introduction to the relevant basics to understand the paper.

1.1 Motivation & Goal

The FMI standard was created to allow the exchange of standardized models within and outside of company boundaries. The topic of Sensitivity Analysis (SA) has already played a role in the development of the standard, for example, to be able to perform uncertainty assessments on models. The groundbreaking developments in the field of machine learning in recent years have led to an increasing fusion of classical engineering disciplines such as scientific computing with machine learning, establishing the research field of *Scientific Machine Learning* (Baker et al. 2019; Rackauckas, Ma, et al. 2021). The backbone of (scientific) machine learning is efficient SA. Therefore, in our view, enhancing SA capabilities for FMUs is a crucial step toward advancing the standard for future applications in research and industry. This improvement will facilitate the use of FMI in machine learning, advanced dynamic optimization, and other fields.

Today, many industrial optimization platforms are used to improve energy efficiency, reduce operational costs, and improve sustainability in various sectors, for example, ABB OPTIMAX@ (Franke et al. 2008) and TLK-Thermo (Petr, Tegethoff, and Köhler 2017). Such platforms often operate based on FMUs because of the seamless integration of physics-based models and efficient handling of complex multi-domain systems. Enabling proper SA in FMUs becomes more and more important, because many industrial optimization problems are highly nonlinear, nonconvex and discontinuous.

While FMI already offers possibilities for SA, for example for integration of FMUs into frameworks for Automatic Differentiation (AD) (s. Sec. 3.4), these are only sufficient to perform SA correctly for continuously differentiable systems. However, for discontinuous systems, further measures are necessary for a correct SA, as we will investigate further. This is of crucial relevance for industrial applications, where most models include discontinuities.

In this position paper, we want to show what is currently possible in FMI and what should be made possible to be open to current and future research fields, e.g., neural FMUs (Thummerer, Mikelsons, and Kircher 2021), the combination of FMUs and Arifical Neural Networks (ANNs). We start by investigating two applications of current research that we believe should be made solvable with FMI. We then examine the current interface offered by FMI and collect requirements with respect to unsolved tasks within these applications. Based on these requirements, we make concrete proposals for an extension of

FMI on the basis of a Layered Standard (LS), i.e. without adjustments to the actual standard release. In parallel, we are already working on a prototype implementation, i.e. the suggestions made are already being tested.

In the following, we provide a brief introduction to the foundational concepts necessary to understand the paper.

1.2 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) is an open standard that defines an interface to exchange dynamic models in the form of a ZIP container which stores the model as a combination of XML files, binaries, and C code. There are three major releases, the most recent version is 3.0.2 (Modelica Association 2024b), while version 2.0.5 might still be the release with the broadest implementation (Modelica Association 2024a). Motivated by different application scenarios for simulation models, FMI supports three different simulation types, while we focus only on the most relevant for SA, which is Model-Exchange (ME) and is further introduced in the next paragraph. Because ME requires external implementation of the solver and event handling routines, incorporation of SA is more straightforward, for example compared to Co-Simulation (CS). Throughout the paper, for readability reasons, we only use the FMI3 command terminology, and are also focusing specifically on the development of the LS for FMI3. However, the proposed adjustments should also be compatible with FMI2 and we try to highlight the necessary adjustments if not.

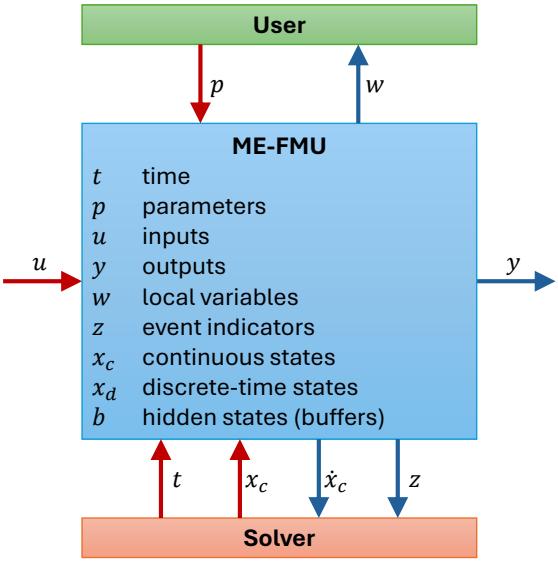

Figure 1. The ME-FMU and its interface signals. Figure adapted from the official standard document (Modelica Association 2024b).

ME-FMUs provide an interface heavily motivated by Ordinary Differential Equations (ODEs) (as further introduced in 1.4) and are therefore paired with ODE solvers to obtain a solution (referred to as *simulation*), see Fig. 1.

The FMI standard facilitates the transfer and exchange of models in simulation tools. However, it has limitations, when the calculation of a model's sensitivity is required.

Layered Standard (LS) The development of the FMI standard is subject to long release cycles to ensure quality and safety of investments, as the standard is widely used in industry. On the other hand, to allow innovation, in FMI 3.0 the concept of Layered Standard (LS) was put in place to be able to introduce new features in an optional and backward compatible way (Bertsch et al. 2023). A LS can add standardized annotations, standardized extra files, and/or new C-API functions.

1.3 Sensitivity Analysis (SA)

Sensitivity Analysis (SA) in the context of FMUs involves studying the influence on the quantities computed by the model (e.g., outputs and state derivatives) w.r.t. variations in quantities that are input to the model (like parameters, states, and inputs). Among others, SA is especially useful for parameter estimation, uncertainty quantification, and optimization.

For example, in optimal chemical production scheduling problems, a mixed-integer linear programming (MILP) solver may be often trapped in suboptimal solutions due to strict reactor start-up time constraints. One can use SA to prioritize the most influential constraints to prevent suboptimal schedules. Moreover, SA can support uncertainty quantification and risk assessment, as industrial systems operate under various uncertainties. By evaluating the range of possible outcomes w.r.t. input variations, SA enhances confidence in optimization results. A further practical example is in hydrogen plant sizing, where SA can assess the impact of load fluctuations, future demand uncertainties, and thermal constraints on transformer efficiency and longevity. This reduces the risk of over- or under-sizing, ensuring that transformers operate within safe margins while optimizing cost and reliability (Gutermuth, Primas, and Bitta 2023).

Last but not least, efficient computation of SA is essential for many modeling and analysis tasks, including hybrid modeling (combining physical and machine learning models) and scientific machine learning. Supporting SA for FMUs is therefore a key enabler for integrating large-scale industrial simulation models into data-driven workflows, es for example required in *neural FMUs* (Thummerer, Mikelsons, and Kircher 2021).

1.4 Ordinary Differential Equations (ODEs)

ME-FMUs without events can be interpreted straightforward as ODEs. The ODE within this work is denoted with

$$\dot{x} = f(x, p, t), \tag{1}$$

where the function f is referred to as *right-hand side*, x the system state, $\dot{x}$ the system state derivative, p parameters and t the time. Note that external inputs to the system could be implemented by augmenting x or p. In this work,

DOI 10.3384/ecp218681

we use this formalism for a simplified description of ME-FMUs without events.

To solve an ODE, a variety of numerical integration schemes exist, that are often incorporated into an abstract definition for arbitrary ODE solvers:

$$x(t) = ODESolve(f, x_0, p, t_0, t), \tag{2}$$

where x_0 and t_0 are the initial state and time to solve the ODE, (Hairer, Nørsett, and Wanner 1992). To obtain a time-discretized solution for f, *ODESolve* can be called recurrently, leading to a *simulation loop* as in Alg. 1.

Algorithm 1 *ODESim*: Simple simulation loop without events.

Input: right-hand side f
Input: time points t_i for $i \in \{0,...,N\}$
Input: initial state x_0, parameters p
$n = 0$ ▷ initialize step counter
while $t_n < t_N$ **do** ▷ final time not reached
  $x_{n+1} = ODESolve(f, x_n, p, t_n, t_{n+1})$
  $n = n + 1$
end while
Output: states $X = \{x_0, ..., x_N\}$

In addition, this concept can be extended to ODEs including *events*. Events are points in time where the system is allowed to change discontinuously. The time point of an event can be known in advance or depend on the system state by defining an event indicator (Modelica Association 2024b). However, this requires a different interface for *ODESolve*, because the termination time for the solver is now not known beforehand (Hairer, Nørsett, and Wanner 1992) and (Chen, Amos, and Nickel 2020):

$$t^*, x(t^*) = ODESolveEvent(f, c, x_0, p, t_0, t). \tag{3}$$

Here, t^* is the event time and the event function c is defined as zero at any event time:

$$c(x(t^*), p, t^*) = 0. \tag{4}$$

This is a simplified view, and in practice a common approach is to monitor changes in signs of a vector containing multiple event indicators, instead of an event function stopping exactly at zero. An event often corresponds to a discontinuous change in the function f. In essence, processing events (so-called *event handling*) captures the following steps: The event time point is determined (by investigation of the event function zero crossing), the solver is stopped, the discontinuous state change is performed (here, by calling the event affect function a) and a fresh integration is started from the new state after the event. The event affect function a computes a new state for after the event $x(t^+)$ based on the quantities at time t^- before the event, and is defined as:

$$x(t^+) = a(x(t^-), p, t^-) = 0. \tag{5}$$

In the actual implementation, it is distinguished between events that solely depend on time (*time events*) or also on states (*state events*). A more detailed introduction on the topic of event handling is not relevant for this work and is omitted for brevity.

Algorithm 2 *ODESimEvent* : Simple simulation loop with events.

Input: right-hand side f
Input: condition c, affect a
Input: time points t_i for $i \in \{0,...,N\}$
Input: initial state x_0, parameters p
$n = 0$ ▷ initialize step counter
$\tilde{t} = t_0$ ▷ init. current time
$\tilde{x} = x_0$ ▷ init. current state
while $t_n < t_N$ **do** ▷ final time not reached
  $\tilde{t}, \tilde{x} = ODESolveEvent(f, c, \tilde{x}, p, \tilde{t}, t_{n+1})$
  while $\tilde{t} \neq t_{n+1}$ **do** ▷ next sol. time point not reached
    $\tilde{x} = a(\tilde{x}, p, \tilde{t})$ ▷ handle event
    $\tilde{t}, \tilde{x} = ODESolveEvent(f, c, \tilde{x}, p, \tilde{t}, t_{n+1})$
  end while
  $x_{n+1} = \tilde{x}$
  $n = n + 1$
end while
Output: states $X = \{x_0, ..., x_N\}$

Further, in the FMI specification, the concept of *superdense* time is applied, allowing events to be ordered even if they are triggered for the same t. For reasons of clarity, this was simplified within this work. In general, the FMI specification (Modelica Association 2024a; Modelica Association 2024b) provides a valuable source for this topic in detail.

2 Related Work

We now turn to related work on SA for FMUs, with a particular focus on discontinuous systems.

FMISensitivity.jl The Julia library *FMISensitivity.jl*[1] implements fully differentiable FMUs using built-in FMI functions as far as available, while sensitivities not considered in FMI are approximated by finite differences. Further, SA relevant to practice, in terms of reasonable computing effort, requires additional but optional FMI features, like `fmi3Get/SetFMUState` for getting and setting the FMU memory state. The implemented workarounds are investigated in more detail in Sec. 3. Before porting the functionality to *FMISensitivity.jl* in 2023, these features were implemented as part of *FMIFlux.jl*.

Time-freezing This approach reformulates non-smooth systems with state jumps into systems that appear smooth from the perspective of a numerical solver (Nurkanović, Sartor, et al. 2021). It introduces a pseudo-time variable and clock state to enforce the discontinuities to lay

[1] `https://github.com/ThummeTo/FMISensitivity.jl`.

in the derivative rater than in the state itself. During the frozen phases, an auxiliary dynamic system evolves the state smoothly in pseudo-time to mimic the effect of the jump. As a result, the original non-smooth problem is more suitable to conventional ODE solvers without adding integer variables. However, constructing such an auxiliary differential equation is not trivial, and introducing pseudo-time variables may increase the computational effort.

Finite Elements with Switch Detection (FESD) This discretization scheme embeds switch detection directly into the time discretization process (Nurkanović, Sperl, et al. 2024). Essentially, it detects state jumps using complementarity constraints and the collocation method. Hence, FESD is able to pose the time points exactly where events happen without relying on the event handling process. The main challenges remain the implementation overhead of a robust switch detection mechanism within a finite element framework and the computational complexity due to the increase of optimization variables and constraints. An implementation of this, as well as the time-freezing approach, can be found as part of the NOSNOC software package (Nurkanović and Diehl 2022).

CasADi In (J. Andersson and Goppert 2024), the integration of SA for discontinuous systems within the *CasADi* tool (J. A. Andersson et al. 2019) is investigated. The paper also gives a rough outlook on how to make discontinuous SA available for FMUs under similar considerations to those we make in the course of this work. However, the authors omitted existing work regarding SA for FMUs: Integrate-then-differentiate, as well as differentiate-then-integrate approach, is available since 2021[2] as part of *FMIFlux.jl* and since 2023 as part of *FMISensitivity.jl*, both providing seamless integration with *SciMLSensitivity.jl* (Rackauckas, Ma, et al. 2021) and *DiffEqCallbacks.jl* (Rackauckas and Nie 2017) that provide implementations for a multitude of SA approaches.

Finite Differences The most obvious approach to SA would be to perform numerical differentiation of the output of *ODESolve*. There are two main disadvantages: a high amount of numerical noise which can be avoided by freezing the step-sizes and number of iterations or alternatively by internally differentiating (Hairer, Nørsett, and Wanner 1992), and that the cost increases linearly with the number of parameters — using adjoint equations avoids this second issue. On the other hand, there is the great advantage that even systems with discontinuities can be analyzed, which is why this method can be found in practical application despite its poor computational performance.

However, all the investigated solutions are suboptimal from a mathematical perspective and are either not generally applicable or scale insufficiently with the size of the problem due to limitations in FMI. This strongly motivates the derivation of an addition to the standard (Layered Standard), that is able to provide access to computational efficient SA, while still being fully backward compatible with FMI.

[2]Experimental 2021, stable release in 2022 (Thummerer, Stoljar, and Mikelsons 2022).

3 Requirements Analysis

In this section, we want to derive requirements for the LS. The essential part of the requirements is the availability of the various Jacobians that are needed for a correct SA. However, it is important to remember that in many cases we are actually interested in *results* of operations *including* the Jacobian, rather than the Jacobian itself. This especially captures Jacobian-Vector Products (JVPs) and Vector-Jacobian Products (VJPs). Note that JVPs are not only useful for SA, but also for solving large ODEs (Ascher and Petzold 1998) with implicit solvers using iterative approaches (e.g., Krylov subspace methods).

We start collecting requirements by investigating two use cases that should enabled with the new LS. Both applications share the question posed by SA, but look at the actual problem from two different perspectives: while a *parameter optimization* task (s. Sec. 3.1) is defined by the problem of identifying sensitivities for variables *within* the FMU, the *neural FMU* approach (s. Sec. 3.2) requires determination of sensitivities for variables *outside* the actual FMU. As a result, different sensitivities are required to solve both tasks.

3.1 Parameter Optimization

One of the most common applications of SA in the field is the optimization of ODE parameters within a model, often referred to as *model calibration*. When optimizing ODEs, an objective l is defined that is either minimized (e.g., *loss* in supervised machine learning) or maximized (e.g., *reward* in reinforcement learning). Typically, this loss is not defined on the right-hand side of the ODE itself, but on the *solution* of the ODE.

The gradient of a loss function defined on the ODE solution X, can be derived by applying the chain rule:

$$\frac{dl(X(p),p)}{dp} = \frac{\partial l(X(p),p)}{\partial X(p)} \frac{\partial X(p)}{\partial p} + \frac{\partial l(X(p),p)}{\partial p}, \quad (6)$$

where the ODE solution X can stem from *ODESim*, as well as *ODESimEvent*.

SA for *ODESim* As can be investigated in algorithm 1, the consecutive evaluation of *ODESolve* leads to the determination of X. On derivative level, this results in a chain of Jacobians containing

$$\frac{\partial x_{n+1}}{\partial x_n} = \frac{\partial ODESolve(f, x_n, p, t_n, t_{n+1})}{\partial x_n}, \quad (7)$$

growing with every step of *ODESolve* that needs to be performed to determine the required Jacobian $\partial X / \partial p$. Most

DOI 10.3384/ecp218681

ODE solvers, for example explicit and implicit Runge-Kutta methods, determine x_{n+1} by one or more evaluations of f. This results in differentiation of f with respect to state and parameters, so $\partial f/\partial x_n$ and $\partial f/\partial p$. These Jacobians can also be found by investigating the sensitivities required for the continuous adjoint method (Céa 1986; Serban and Hindmarsh 2003; Sapienza et al. 2024). If the right-hand side depends on time, further the time gradient $\partial f/\partial t$ is required. Of course, to obtain a complete derivative chain, also the arithmetic operations inside the ODE solver need to be differentiated. However, the detailed chain of Jacobians depends on the type and implementation of the numerical solver and is therefore omitted.

SA for *ODESimEvent* Very similar to *ODESim*, sensitivities for *ODESimEvent* can be collected by investigation of algorithm 2. However, there are differences: First, *ODESimEvent* also determines the event time t^* besides the system state, therefore differentiation over *ODESolveEvent* results in determination of the Jacobian $\partial x^*/\partial x_n$ and the additional time gradient $\partial t^*/\partial x_n$. Second, the event time t^* depends on the zero crossing of the event condition c, which involves differentiation of the event condition with respect to its arguments, so $\partial c/\partial x_n$ and $\partial c/\partial t$. These sensitivities can also be found by investigating the continuous adjoint method for event-ODEs (Chen, Amos, and Nickel 2020; Pfeiffer 2008). Finally, also the derivative of the new state after the event is needed by differentiation of the *affect* function a, in particular the Jacobian of the state after the event w.r.t. to the state before the event $\partial x^+/\partial x^-$.

SA for calculated Parameters In practice there are often also calculated parameters $b = g(p)$ that are computed at the start of the simulation. These calculations may be trivial (e.g., calculating the mass based on the density and volume parameters), or complicated, as, for example, calculating polynomial coefficients based on curve fitting. The set of parameters can be extended to calculated parameters to accumulate sensitivities with respect to b and then only compute the VJPs for $\partial g/\partial p$ once. Differentiating these functions should be straightforward even if they are not needed for analytic Jacobians. Parameter subexpressions can be treated together with calculated parameters, as we are not interested in individual calculated parameters but only in their overall impact.

3.2 Neural FMUs

In (Thummerer, Mikelsons, and Kircher 2021), neural FMUs where introduced as the combination of FMUs, ANNs and a numerical ODE solver. In the easiest case, only a single FMU and a single ANN are connected. Even if a common scenario is the modification of the FMU state derivative by the ANN, a variety of different combination schemes are possible. Whereas neural FMUs are in general not restricted to ME-FMUs, we only focus on ME within this work, because we see this as more relevant for the topic of proper SA, and it is definitely more relevant within the topic of neural FMUs. However, we give a short outlook on CS in the conclusion section.

As introduced, within neural FMUs, no restrictions are made with respect to the connections between the FMU and ANN(s). To make this more tangible, in Fig. 2 all possible positions to extend a ME-FMU by ANNs are given. In addition to the theoretical possibility of doing this, there is of course also the question of practical relevance, which we shall examine below by a brief investigation of fields of applications:

- One of the first applications of hybrid modeling (in terms of combining physical and machine learning models) was the **correction of parameters** by an ANN (Psichogios and Ungar 1992). This is still a meaningful measure within similar applications.
- The most valuable point of intervention is the **correction of state derivatives**. Note that this allows, for example, to incorporate additional forces (like friction) to a system because forces act on the state derivative (acceleration). Manipulation of the state derivative allows us to change the entire behavior of the system because it allows us to overwrite or extend the actual right-hand side of the ME-FMU.
- Time can be manipulated to induce, e.g., **time shifts** to compensate for unsynchronized measurements. In the parallel contribution (Thummerer, Jarmolowitz, et al. 2025), this is investigated.
- By changing the **system input**, corrections can be made to faulty signals. This further opens up the topic of learning an input trajectory to achieve *optimal control*.
- If only the **system outputs** (or local variables) are changed, the correction is learned on top of the simulation performed, without actually influencing the ODE solution, leading in general to less generalization within the ANN. However, this can still be useful in applications where generalizability is less important.
- Manipulation of event indicators allows, for example, for **augmentation of the event indicators** to learn for not modeled event conditions and therefore new discontinuities.
- **Correcting the system state** before passing it to the FMU allows for correcting modeling errors like general shifts in state-space (Thummerer, Mikelsons, and Kircher 2021) or equilibrium position for fluid simulations such as in (Thummerer, Tintenherr, and Mikelsons 2021).

Based on the finding that such architectures have practical relevance, the question arises as to which sensitivities are required for optimization including such architectures.

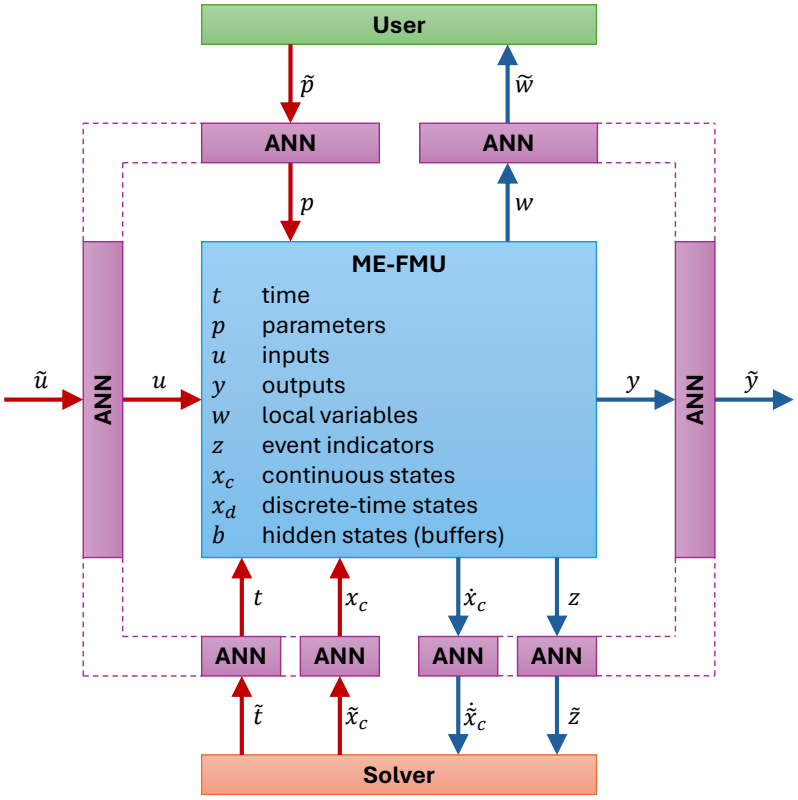

Figure 2. The neural FMU (ME), showing possible intervention points where ANNs can be incorporated to extend the ME-FMU. Individual ANNs could be used for each signal, or all causal input ANNs (p, u, t and x_c) could be merged to obtain a single input ANN (purple, dashed silhouettes). The same applies for the causal output ANNs (w, y, $\dot{x}_c$ and z). The discrete-time state x_d and the buffer b are not accessible from outside the FMU within the standard definition, granted access to x_d is part of the LS-SA, however.

Since all possible permutations of existing ANNs are to be investigated, it is expedient to assume the most complex case: All ANNs are incorporated.

3.3 Required Sensitivities

Based on the two fields of applications introduced and the corresponding observations, all necessary sensitivities are collected below. It is required to allow for JVP and VJP operations involving the Jacobians[3] $\partial\gamma/\partial\upsilon$, where $\upsilon \in \{t, x_c, u, p\}$ and $\gamma \in \{\dot{x}_c, z, y, w\}$. Furthermore, the actual event affect function that computes the new state of the system after event x_c^+ based on the system state before the event x_c^- is required but neglected in the picture to be compatible with the illustration in the standard specification. Furthermore, sensitivities are required not only for the continuous state, but also for the entire state of the system, which additionally requires the discrete state x_d. In summary, the required sensitivities are $\upsilon \in \{t, x_c^-, x_d^-, u, p\}$ and $\gamma \in \{\dot{x}_c, x_c^+, x_d^+, z, y, w\}$.

3.4 Available methods within FMI

In FMI, there are two commands available that allow to access partial derivatives over an FMU. First, `fmi3GetDirectionalDerivative` allows for JVP-multiplication with a custom seed vector, while `fmi3GetAdjointDerivative` provides functionality for VJP-multiplication. Although this does not allow one to directly access the Jacobians, it allows a straightforward integration within AD frameworks. Forward-mode AD involves consecutive evaluations of JVPs, which allows for a natural propagation of the current sensitivity seed through the FMU with `fmi3GetDirectionalDerivative`. The other way round, for reverse-mode AD, consecutive VJPs are computed and `fmi3GetAdjointDerivative` provides an interface for backpropagation of the adjoint sensitivity (or co-tangent) through the FMU. However, there are two issues with these commands: First, the implementation is optional, and, therefore, only a fraction of tools support these commands. Second, and even more important, not all required sensitivities (value references) are accessible by these commands, event if they are implemented. This is investigated in detail, along with solution proposals, in the next section.

[3] For the case of $\gamma = t$ (scalar time) it is more common to refer to a *gradient* than *Jacobian*.

4 Layered Standard

In the following subsections, we investigate all open issues derived from the introduced applications. For each issue, we will highlight the problem and available workarounds.

DOI
10.3384/ecp218681

Workarounds already implemented within *FMISensitivity.jl* and *FMIFlux.jl* are briefly described below. Finally, possible solutions and their technical implementation in the form of a LS are investigated. We explicitly try to reuse the existing command interface as much as possible, as we want to minimize the implementation effort in the tools and enable a wide adaptation of the LS-SA.

4.1 Event condition

The FMI standard allows for the calculation of derivatives of state derivatives $\dot{x}$ and outputs y with respect to states x and inputs u. If the system is subject to events, a complete SA requires the sensitivity of the point in time where an event occurs. As indicated in subsection 3.1, this requires derivatives of the event condition, or as it is named in FMI, the event indicator. Since FMI3, event indicators are part of the model variables (so they are exposed), which was not the case for FMI2. Within the FMI specification, there is no restriction that prohibits providing sensitivities for event indicators via the directional or adjoint derivative commands, however it is also not enforced.

Workaround The only generally applicable workaround to obtain partial derivatives w.r.t. event indicators is to sample them via finite differences. This way it is implemented in *FMISensitivity.jl*. The required number of samples scales linearly with the Jacobian size to multiply with, that is, linearly with the number of rows for forward AD or the number of columns for reverse AD.

Proposed Solution (LS-SA) The proposed solution for the LS-SA is straight-forward: Because the model variables of the event indicators are already known in FMI3, it is only necessary to provide sensitivities for the event indicators via `fmi3GetDirectionalDerivative` and `fmi3GetAdjointDerivative`. For FMI2, it is additionally required to expose the event indicators in the same way as they are exposed in FMI3, by adding the corresponding entries to the section `<ModelStructure>` in the model description.

4.2 Discontinuous State Change

After the event is handled, the numerical integration is restarted with a new state x^+. However, also this new state may depend on the state before the event x^-, time, inputs before the event and/or parameters. These sensitivities are not present within FMI — in general, it is not possible to access sensitivities w.r.t. to quantities *before* an event after event handling was performed.

Workaround One could sample the required sensitivity by restarting the simulation and disturbing the investigated variable right before the event (finite differences). This is computationally infeasible, because this requires a new simulation for every required sensitive state / input / parameter.

In *FMIFlux.jl*, a much more performant approach is implemented, however, this requires the optional FMI features `fmi3Get/SetFMUState`. If these commands are available, a memory snapshot of the FMU can be made right before the event. In this way, sampling becomes much more efficient because the snapshot can be used to reinitialize right before the event to disturb the signals for sampling, instead of starting a new simulation. In this way, the complexity reduces to only performing one evaluation of the system of equations rather than performing an entire simulation.

Proposed Solution (LS-SA) The challenge is that the x^+ and x^-, which are required for the Jacobian $\partial x^+ / \partial x^-$, represent the same state (model variable) at different locations in (super-dense) time. In FMI, there is no mechanism to access partial derivatives w.r.t to the same variable at different points in super-dense time, because the state before and after the event have the same value reference.

However, in the Modelica modeling language, previous values before event handling can be accessed using the function `pre`. We therefore propose to introduce new model variables that can be used to access previous values of states. This captures the continuous, as well as the discrete-time state. Because derivatives of continuous states are introduced with the `derivative` keyword within the definition of the model variable, we propose a new keyword `previous` for previous states before the event. A code snippet for this is given in Listing 1.

Listing 1. Example for a state definition within the model description, including the new *previous* attribute.

```
<Float64
    name="mass.s"
    valueReference="33554432"
    description="absolute position of mass"
    unit="m"
</Float64>
<Float64
    name="der(mass.s)"
    valueReference="587202560"
    description="derivative of mass.s"
    unit="m/s"
    derivative="33554432">
</Float64>
<Float64
    name="pre(mass.s)"
    valueReference="587202561"
    description="previous value for mass.s"
    unit="m"
    previous="33554432">
</Float64>
```

In case of multiple events occurring at the same point in time, the `previous` model variable actually refers to the *previous* value before the last event handled, and not the value before the first event at the time instant. Finally, note that this measure is not necessary for other relevant inputs like u and p, because they do not change in super-dense time, for example $\partial x^+ / \partial p^- = \partial x^+ / \partial p^+$.

4.3 Time Gradients

As investigated, the gradient with respect to time is needed. Since FMI3, time (or the independent variable

in general) is registered as special model variable with `causality="independent"`. However, there is no enforced possibility within FMI to request partial derivatives w.r.t. time.

Workaround As for the event condition, the influence of time, e.g. on the state derivative or outputs, can be sampled straightforwardly during continuous time by disturbing only the time and re-evaluation of the FMU.

Proposed Solution (LS-SA) We would like to enforce that directional and adjoint derivatives are allowed to be requested w.r.t. the independent variable during continuous time.

4.4 Event Time

In addition to event indicators that depend on state and time, events can be registered solely depending on time. The time of these events is known exactly, but only stepwise, the next event time is determined during initialization (the first time event) or during event handling (for the upcoming time events). Such events are not determined by zero crossings of event indicators, hence their influence on the FMU's outputs has to be determined separately.

Workaround To calculate the influence on the current state or parameter on the next event time, a sampling-based approach is used as for the state change in the previous Sec. 4.2: Before actually handling the current time event, a memory snapshot (`fmi3GetFMUState`) is performed, and the influence of the state and parameters can be successively sampled by disturbing one input to SA, handling the time event and quantifying the influence on the determined event time. This requires at least one sample (one event handling procedure) for every state and parameter in the system and is therefore expensive for large simulations.

Proposed Solution (LS-SA) As outlined, the next event time may also depend on the current time. Therefore, it is not sufficient to require the time to be defined as model variable (cf. Sec. 4.3), but necessary to require an additional one for the *next* event time. This enables to calculate the sensitivity of the next event time w.r.t. state and parameters as well as the influence of the current event time on the next event time. Although these sensitivities are required for SA at the next event, they already have to be calculated at the current event and need to be cached.

4.5 Parameter Sensitivities

Even if FMI supports functions to obtain directional derivatives, it is important to state that parameter sensitivities are not available during the actual simulation. Even tunable parameters are only available in event mode (Modelica Association 2024b), which is not useful during continuous-time simulation. This is of course a huge problem and actively prevents parameter SA for FMUs. There are also good reasons to not treat these parameters as tunable, as we are often (e.g., in case of parameter SA) considering the impact for the entire simulation. Furthermore, tunable parameters can complicate the storage of the memory state, and SA requires storing a large number of memory states.

Workaround To our knowledge, there is no obvious workaround to this. However, we observed that some FMU implementations do not implement this limitation and provide correct parameter sensitivities even during continuous-time mode.

Proposed Solution (LS-SA) The proposed solution is straightforward: Within the LS, we enforce that parameter sensitivities (at least for tunable parameters) must be accessible via the already existing interface for directional and adjoint derivatives.

Beyond the listed issues, we would also like to add requirements for the application of neural FMUs in the field. In addition to the mathematical considerations for a comprehensive SA, the optimization or training process can be made more efficient and robust with the following enhancements.

4.6 Get/Set Discrete State

A common technique in the training of neural FMUs, and in machine learning in general, is mini-batching. Training data is divided into chunks called *mini-batches*, and parameter gradients are determined for each chunk rather than the entire dataset. A new batch is selected for each training step, either algorithmically or randomly.

If mini-batching is paired with neural FMUs, or mixed continuous-discrete systems in general, a new challenge pops up: The system needs to be initialized for the initial state of every mini-batch. This covers the continuous state, but also the discrete-time state of the FMU. Note that the discrete state can influence the system dynamics, in the extreme case, the discrete state can be used to switch between completely different sets of equations for the continuous right-hand side. In this case, the discrete state is sometimes referred to as *mode* of the system.

By default, FMI completely hides the discrete-time state of the FMU. It is neither readable nor writable from the outside. However, this is required to initialize the system based on data. However, it is important to state that for proper initialization the continuous and discrete states must *match*. Depending on the system model, not every combination of continuous and discrete states is meaningful and leads to a solvable system.

Workaround For now, FMI does not provide an interface to directly access the discrete state, but an interface to catch it *indirectly*. The corresponding commands are `fmi3Get/SetFMUState`, which create (or activate) a memory copy of the entire FMU memory state, including the continuous and discrete state, but also additional variables, such as starting values for iteration of algebraic loops. It is also important that commands for the memory

DOI
10.3384/ecp218681

states are declared *optional* in the FMI specification and only a fraction of tools implemented these.

Furthermore, while these commands can be used to *revisit* a discrete state after capturing it once, they do not allow one to actually modify the discrete state. For example, if the system needs to be initialized in a very specific discrete state, one needs to *guide* the system to this specific state, e.g. by controlling inputs, in order to capture the intended discrete state. This is an immense effort and is generally not feasible in the case of a complex simulation model. This workaround should therefore only be regarded as a makeshift solution.

Proposed Solution (LS-SA) Even if this step is associated with technical challenges, we see no alternative to allowing reading / writing of the discrete state of the system, as is allowed for the continuous part of the state within FMI. We propose the following additions to FMI:

1. Because the data type of the discrete state may vary, compared to the continuous state which is always `Float64` / `Float32`, we do not suggest introducing new commands such as `fmi3Get/SetDiscreteState`. We propose to reuse the existing commands for reading/writing variables of the FMU, like e.g. `fmi3Get/SetInt32`. This way, no new commands are introduced, which we see as the main obstacle when implementing a LS, and there is no problem with multiple data types used for the discrete state, as it can be set/get by multiple consecutive calls (e.g. mixing boolean and integer values).

2. The value references of the discrete states must be made available from outside the FMU, in order to allow for setting/getting the discrete state. We propose to add this information to the model description, as part of the section `<ModelStructure>`. There, we like to add a new section `<DiscreteState>`, where value references of discrete states are listed in the same way as, e.g. for outputs. Because FMI does not allow us to add new fields to the standardized model description, the addition of fields does not happen within the existing model description. Technically speaking, a separate file is created consisting only of the new fields and a minimum representation of the model description XML tree around.

3. Discrete states are only required to be settable in event mode (where the discrete state changes during regular simulation), and we propose to require triggering event handling consecutively to ensure that the given continuous states match the discrete states.

4.7 Comply with Assertions

The use of assertions when modeling large systems is a meaningful measure in developing correct and easily maintainable simulation models. Typically, assertions are formulated in binary form, i.e. they remain unnoticed until a condition is violated. The existence of an assertion and the monitored condition cannot be seen outside the FMU. This has drastic effects on the application of hybrid modeling with FMUs (e.g. neural FMUs). If, for example, the system dynamics is changed by an ANN, the violation of a condition within an assertion can only be detected when it is already too late and an exception is thrown, which often leads to the termination of the simulation and thus the SA.

To our knowledge, there is **no workaround** to this.

Proposed Solution (LS-SA) We propose to introduce a concept related to the event indicator interface, which we call straightforward *error indicators*. Error indicators provide a distance measure for the violation of modeling errors. Unlike event indicators, the sign is important. Negative error indicators identify the non-existence of a modeling error, whereas positive error indicators show that a modeling error occurred. For the case of at least one positive error indicator, we do not expect the FMU to provide meaningful values, e.g. returning `fmi3StatusError` or `fmi3StatusDiscard` for calls to `fmi3GetX`. To implement error indicators, we propose the following modifications:

1. Error indicators are introduced as model variables in the same way as event indicators, and are made public in the section `<ModelStructure>` in the model description, where we add a new subsection `<ErrorIndicator>`, again in the same way as for `<EventIndicator>`. As for the discrete state in 4.6, this measure is implemented in a parallel model description file and not by modification of the standardized FMI model description.

2. To make the values of error indicators accessible, two obvious strategies are available. First, the error indicators could be accessed through `fmi3GetFloat64` as any other (`Float64`) model variable. For now, we see no reason to not enforce all error indicators to be of type `Float64` (for 64-bit FMUs). Second, because all event indicators share the same data type, a new command could be introduced in analogy to `fmi3GetEventIndicators`, see code 2, that allows capturing all error indicators at once.

3. As proposed for event indicators, we also like to keep error indicators differentiable. Although this is not required for proper SA, this opens up an interesting and valuable new use case: Explicit regularization. If not only the distance to an error is known, but also in which direction parameter changes influence the error (*gradient*), error indicators can be used to actively prevent errors during training, for example by adding the (differentiable) error indicators to the training objective as additional summands (referred to as *regu-*

larization terms). Sensitivities could be accessed in analogy to the event indicators.

Listing 2. The new error indicator function.

```
typedef fmi3Status fmi3GetErrorIndicators(
    fmi3Instance instance,
    fmi3Float64 errorIndicators[],
    size_t nErrorIndicators);
```

Although we consider all the proposed measures within this section important, we have sorted them in descending priority order (with the aim of carrying out a correct SA) in order to support a targeted implementation.

5 Conclusion

In summary, we have investigated the opportunities that already exist for SA in FMI. By investigating two example applications (parameter SA and neural FMUs), we were able to identify deficits, particularly in the analysis of discontinuous systems. We went on to consider how we could close these gaps (mathematically and technically) and made a concrete proposal for the LS-SA, which will be able to solve the issues investigated under reasonable computational performance.

Current & Future Work Even if the LS is not yet fully implemented, a dedicated working group within the research project *OpenSCALING*, including tool vendors, users, and academia, continues to develop and implement the LS to be able to publish it during the project period. As soon as more prototypes of tool implementations are available, we want to examine and quantify the practical computational benefits of the LS besides the theoretical considerations.

One major part of future work is a more detailed investigation and specification of the proposed measures, e.g. it is necessary to specify allowed (or forbidden) attributes for the newly introduced model variables.

Of course, the development of LS-SA is also strongly driven by *specific* applications for which we are aiming, which will benefit greatly from the realization of the standard extension. For example, training of neural FMUs, the combination of ANNs and physics-based simulation models captured in an FMU, will benefit from a significantly faster training process (multiple factors, depending on the number of states and events).

In theory, the proposed LS-SA can be extended for CS-FMUs. However, this introduces additional challenges because parts of SA, which are now part of the simulator, need to be incorporated to the FMU. Further, a clever interface needs to be discussed that allows for efficient determination of sensitivities, for example by defining required sensitivities before starting the actual simulation (e.g. during the setup of experiment).

This publication lays the methodological foundation for further discussions on a large and importing topic within the standard, as well as prototype implementations within importing and exporting tools. It is important for us to point out that the LS is in active development, but not officially released. We would like to draw the community's attention to the development, as well as the intended measures, and we hope to achieve a broad discussion of this topic in the community so that a strong LS with broad adoption can emerge from this. We welcome any kind of participation.

Funding

This research was partially funded by ITEA4-Project OpenSCALING (Open standards for SCALable virtual engineerING and operation) No. 22013. For more information, see: `https://openscaling.org/` (accessed on 30 April 2025).

Author Contributions

Conceptualization, T.T., O.H., S.C., G.J., B.T., M.L.; methodology, T.T., O.H., G.J., B.T., M.L., S.C.; writing—original draft preparation, T.T., O.H., S.C., G.J., B.T., M.L.; writing—review and editing, T.T., O.H., S.C., G.J., M.L.; visualization, T.T.; All authors have read and agreed to the published version of the manuscript.

Abbreviations

AD Automatic Differentiation

ANN Arifical Neural Network

CS Co-Simulation

FMI Functional Mock-up Interface

LS Layered Standard

FESD Finite Elements with Switch Detection

FMU Functional Mock-up Unit

JVP Jacobian-Vector Product

ME Model-Exchange

MILP mixed-integer linear programming

ODE Ordinary Differential Equation

SA Sensitivity Analysis

VJP Vector-Jacobian Product

References

Andersson, Joel and James Goppert (2024). "Event support for simulation and sensitivity analysis in CasADi for use with Modelica and FMI". In: *Modelica Conferences*, pp. 99–108.

Andersson, Joel AE et al. (2019). "CasADi: a software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11, pp. 1–36.

Ascher, Uri M. and Linda R. Petzold (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia, PA: Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611971392.

Baker, Nathan et al. (2019). *Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence*. Tech. rep. USDOE Office of Science (SC), Washington, DC (United States).

Bertsch, Christian et al. (2023). "Beyond FMI-Towards New Applications with Layered Standards". In: *Modelica Conferences*, pp. 381–388.

Céa, Jean (1986). "Conception optimale ou identification de formes, calcul rapide de la dérivée directionnelle de la fonction coût". In: *Revue Française d'Automatique, Informatique, et Recherche Opérationnelle (RAIRO)* 20.4, pp. 371–402.

Chen, Ricky T. Q., Brandon Amos, and Maximilian Nickel (2020). "Learning Neural Event Functions for Ordinary Differential Equations". In: *CoRR* abs/2011.03902. arXiv: 2011.03902. URL: https://arxiv.org/abs/2011.03902.

Franke, Rüdiger et al. (2008). "Model-based online applications in the ABB Dynamic Optimization framework". In: *Proc. of the 6th Int. Modelica Conference, Bielefeld. www. modelica. org/events/-modelica2008/Proceedings/sessions/session3b1. pdf*.

Gutermuth, Georg, Bernhard Primas, and Jan Bitta (2023). "Sizing matters". In: *ABB Review*. URL: https://new.abb.com/news/detail/103158/sizing-matters.

Hairer, E., S.P. Nørsett, and G. Wanner (1992). *Solving Ordinary Differential Equations I Nonstiff problems*. Second. Berlin: Springer.

Modelica Association (2024a-11). *Functional Mock-up Interface for Model Exchange and Co-Simulation. Document version: 2.0.5*. Tech. rep. Linköping: Modelica Association. URL: https://github.com/modelica/fmi-standard/releases/download/v2.0.5/FMI-Specification-2.0.5.pdf.

Modelica Association (2024b-11). *Functional Mock-up Interface Specification. Version 3.0.2*. Tech. rep. Modelica Association. URL: https://fmi-standard.org/docs/3.0.2/.

Nurkanović, Armin and Moritz Diehl (2022). "NOSNOC: A Software Package for Numerical Optimal Control of Nonsmooth Systems". In: *IEEE Control Systems Letters* 6, pp. 3110–3115. DOI: 10.1109/LCSYS.2022.3181800.

Nurkanović, Armin, Tommaso Sartor, et al. (2021). "A Time-Freezing Approach for Numerical Optimal Control of Nonsmooth Differential Equations With State Jumps". In: *IEEE Control Systems Letters* 5.2, pp. 439–444. DOI: 10.1109/LCSYS.2020.3003419.

Nurkanović, Armin, Mario Sperl, et al. (2024). "Finite elements with switch detection for direct optimal control of nonsmooth systems". In: *Numerische Mathematik* 156.3, pp. 1115–1162.

Petr, Philipp, Wilhelm Tegethoff, and Jürgen Köhler (2017). "Method for designing waste heat recovery systems (WHRS) in vehicles considering optimal control". In: *Energy Procedia* 129, pp. 232–239.

Pfeiffer, Andreas (2008). "Numerische Sensitivitätsanalyse unstetiger multidisziplinärer Modelle mit Anwendungen in der gradientenbasierten Optimierung". PhD thesis. Martin-Luther Universität Halle-Wittenberg.

Psichogios, Dimitris C. and Lyle H. Ungar (1992). "A hybrid neural network-first principles approach to process modeling". In: *AIChE Journal* 38.10, pp. 1499–1511. DOI: https://doi.org/10.1002/aic.690381003.

Rackauckas, Christopher, Yingbo Ma, et al. (2021). *Universal Differential Equations for Scientific Machine Learning*. arXiv: 2001.04385 [cs.LG].

Rackauckas, Christopher and Qing Nie (2017). "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *The Journal of Open Research Software* 5.1. DOI: 10.5334/jors.151.

Sapienza, Facundo et al. (2024). *Differentiable Programming for Differential Equations: A Review*. arXiv: 2406.09699 [math.NA]. URL: https://arxiv.org/abs/2406.09699.

Serban, Radu and Alan C Hindmarsh (2003). *CVODES: An ODE solver with sensitivity analysis capabilities*. Tech. rep. Citeseer.

Thummerer, Tobias, Fabian Jarmolowitz, et al. (2025). "Br(e)aking the boundaries of physical simulation models: Neural Functional Mock-up Units for Modeling the Automotive Braking System". In: *16th International Modelica & FMI Conference, Lucerne, Switzerland, Sep 8-10, 2025*. (accepted).

Thummerer, Tobias, Lars Mikelsons, and Josef Kircher (2021). "NeuralFMU: towards structural integration of FMUs into neural networks". In: *Proceedings of 14th Modelica Conference 2021, Linköping, Sweden, September 20-24, 2021*. Ed. by Martin Sjölund et al. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181297.

Thummerer, Tobias, Johannes Stoljar, and Lars Mikelsons (2022). "NeuralFMU: Presenting a Workflow for Integrating Hybrid NeuralODEs into Real-World Applications". In: *Electronics* 11.19. ISSN: 2079-9292. DOI: 10.3390/electronics11193202. URL: https://www.mdpi.com/2079-9292/11/19/3202.

Thummerer, Tobias, Johannes Tintenherr, and Lars Mikelsons (2021-11). "Hybrid modeling of the human cardiovascular system using NeuralFMUs". In: *Journal of Physics: Conference Series* 2090.1, p. 012155. DOI: 10.1088/1742-6596/2090/1/012155.