An FMI-Based Initialization Plugin for INTO-CPS Maestro 2

Simon Thrane Hansen(✉), Casper Thule, and Cláudio Gomes

DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark
{sth,casper.thule,claudio.gomes}@eng.au.dk

Abstract. The accuracy of the result of a co-simulation is dependent on the correct initialization of all the simulation units. In this work, we consider co-simulation where the simulation units are described as Functional Mock-up Units (FMU). The Functional Mock-up Interface (FMI) specification specifies constraints to the initialization of variables in the scope of a single FMU. However, it does not consider the initialization of interconnected variables between instances of FMUs. Such interconnected variables place particular constraints on the initialization order of the FMUs.

The approach taken to calculate a correct initialization order is based on predicates from the FMI specification and the topological ordering of both internal connections and interconnected variables. The approach supports the initialization of co-simulation scenarios containing algebraic loops using fixed point iteration. The approach has been realized as a plugin for the open-source INTO-CPS Maestro 2 Co-simulation framework. It has been tested for various scenarios and compared to an existing *Initializer* that has been validated through academic and industrial application.

Keywords: Co-simulation · Initialization · Algebraic loop · Topological ordering · FMI

1 Introduction

Cyber-physical systems (CPS) are becoming ever more sophisticated, while market pressure shortens the available development time. One of the tools to manage the increasing complexity of such systems is co-simulation since it tackles their heterogeneous nature. Co-simulation is a technique to combine multiple black-box simulation units to compute the combined models' behavior as a discrete trace (see, e.g., [12,14]). The simulation units, often developed independently from each other, are coupled using a master algorithm, often developed independently, that communicates with each simulation unit via its interface. This

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University. Finally, we thank the reviewers for the thorough feedback.

© Springer Nature Switzerland AG 2021
L. Cleophas and M. Massink (Eds.): SEFM 2020 Workshops, LNCS 12524, pp. 295–310, 2021.
https://doi.org/10.1007/978-3-030-67220-1_22

interface comprises functions for setting/getting inputs/outputs and computing the associated model behavior over a given time interval. The Functional Mock-up Interface (FMI) standard [4,7] is such an interface prescribing how to communicate with each simulation unit. The interface is used to connect different simulation units, called Functional Mock-up Units (FMUs), exchange values between them, and make them progress in time.

A typical co-simulation consists of three phases: initialization, simulation, and teardown [22]. This work concentrates on the first. The FMI standard specifies criteria for how a single FMU shall be initialized. However, FMI is not concerned with how a connected system of multiple FMUs is initialized correctly as a whole.

The way a system of multiple FMUs should be initialized and interacted with depends on each FMU's implementation and interconnections to other FMUs [9], since these place precedence constraints between the FMU variables. These precedence constraints can introduce algebraic loops between the FMU variables. An algebraic loop places particular requirements on the strategy for both the order of initialization and the method used to calculate the correct initial values of the variables in the algebraic loop [3]. Algebraic loops occur whenever an interconnected FMU variable indirectly depends on itself. Not solving an algebraic loop can lead to a prohibitively high error in the co-simulation result [2], and invalid results, as shown in Sect. 4. It is crucial for all interconnected variables that the initialization procedure ensures that a variable is never read before it is set. For variables within an algebraic loop, the initialization must ensure that all initial values have converged to a fixed point before entering the next phase of the co-simulation.

Other approaches for the generation of co-simulation algorithms have avoided co-simulation scenarios containing algebraic loops since their presence reduces the chance of obtaining a deterministic co-simulation result[1,5,10]. This choice is driven by the fact that not all co-simulation scenarios containing algebraic loops are valid since those algebraic loops never converge, or might converge to unexpected solutions. However, as shown in Sect. 4, solving algebraic loops can be essential to obtaining valid simulation results, and a well-established co-simulation framework should be able to handle these scenarios.

Contribution: This paper describes an approach for calculating the initialization order of an FMI-based co-simulation in linear time of the number of interconnected variables, even when algebraic loops are present. The approach does not put any constraints on the master algorithm chosen to carry out the simulation. The approach is realized as a plugin to the co-simulation framework called INTO-CPS Maestro 2 (Maestro 2), introduced in [22]. The realized plugin has been tested for various co-simulation scenarios and compared to an existing Initializer that has been validated through academic and industrial applications. Furthermore, the calculated initialization order is systematically verified by the semantics of co-simulation introduced in [9,10].

Structure: The paper is structured as follows: Sect. 2 gives a brief background of the formalization of FMUs and Maestro 2. Section 3 describes the approach taken to calculate the initialization order. It is followed by Sect. 5, where the realization

of the approach is presented. Finally, Sect. 7 provides concluding remarks and describes future work.

2 Background

In this section, we provide a formalization of FMI co-simulation and a brief background on INTO-CPS Maestro 2.

2.1 FMU Definitions

To describe the formalization of FMUs, we adopt the vocabulary from [9]. The main definitions of relevance to this paper will be presented, but readers are referred to the original publications for more information. This paper is only concerned with the initialization-phase of a co-simulation, making time of an FMU irrelevant. The formalization from Gomes et al.[9] is extended with new definitions regarding algebraic loops, and convergence of fixed point iteration.

Definition 1 (FMU). *An FMU with identifier c is represented by the tuple*

$$\langle S_c, U_c, Y_c, \mathtt{set}_c, \mathtt{get}_c \rangle ,$$

where: S_c represents the state space of FMU c; U_c and Y_c the set of input and output variables, respectively; $\mathtt{set}_c : S_c \times U_c \times \mathcal{V} \to S_c$ and $\mathtt{get}_c : S_c \times Y_c \to \mathcal{V}$ are functions to set the inputs and get the outputs, respectively (we abstract the set of values that each input/output variable can take as $\mathcal{V}$).

Definition 2 (Scenario). *A scenario is a structure $\langle C, L \rangle$ where each identifier $c \in C$ is associated with an FMU, as defined in Definition 1, and $L(u) = y$ means that the output y is connected to input u. Let $U = \bigcup_{c \in C} U_c$ and $Y = \bigcup_{c \in C} Y_c$, then $L : U \to Y$.*

Note a single output can connect to multiple inputs, but a single input can only rely on a single output. The following definitions correspond to the operations that are permitted in the initialization phase of a co-simulation.

Definition 3 (Output Computation). *The $\mathtt{get}_c(\_, y_c)$ represents the calculation of output y_c of $c \in C$. Given a co-simulation state, it checks whether all inputs that feed-through to y_c are defined.*

Definition 4 (Input Computation). *The $\mathtt{set}_c(\_, u_c, v)$ represents the setting of input u_c of $c \in C$. Given a co-simulation state, it checks whether all outputs connected to u_c are defined.*

Definition 5 (Fixed Point). *The $\mathtt{fixedpoint}_l$ represents an ordered sequence of the setting or getting of all variables of a given SCC l, see Definition 9 for a definition of SCC. The $\mathtt{fixedpoint}_l \subseteq \bigcup_{c \in C} \{\mathtt{get}_c, \mathtt{set}_c\}$*

Definition 6 (Initialization). *Given a scenario $\langle C, L \rangle$, we define the initialization procedure $(I_i)_{i \in \mathbb{N}}$ as is a finite ordered sequence of FMU function calls that needs to be performed in the initialization of a co-simulation scenario. The ordered sequence is defined as: $(f_i)_{i \in \mathbb{N}} = f_0, f_1, \ldots$ with $f_i \in I = \bigcup_{l \in loops} \texttt{fixedpoint}_l$, and i denoting the order of the function call. Loops is defined as the set of all SCC see Definition 9.*

It should be noted that a trivial SCC (see Definition 9) is only a single get or set action and is not regarded as a fixed point outside Definition 6, but just a simple computation.

Definition 7 (Feed-through). *The input $u_c \in U_c$ feeds through to output $y_c \in Y_c$, that is, $(u_c, y_c) \in D_c$, when there exists $v_1, v_2 \in \mathcal{V}$ and $s_c \in S_c$, such that $\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c)$.*

A graph of the dependencies of a co-simulation scenario is established from the interconnected variables by Definition 8. The graph is the foundation for the calculation of the initialization procedure and is therefore referred to as the Initialization Graph. The graph construction is similar to the one in [10], except the later focuses on a general co-simulation step, while this work focus on the initialization phase.

Definition 8 (Initialization Graph). *Given a co-simulation scenario $\langle C, L \rangle$, and a set of feed-through dependencies $\bigcup_{c \in C} \{D_c\}$, we define the Initialization Graph where each node represents a port $y_c \in Y_c$ or $u_c \in U_c$ of some fmu $c \in C$. The edges are created according to the following rules:*

1. *For each $c \in C$ and $u_c \in U_c$, if $L(u_c) = y_d$, add an edge $y_d \rightarrow u_c$ (output to input).*
2. *For each $c \in C$ and $(u_c, y_c) \in D_c$, add an edge $u_c \rightarrow y_c$ (input to output).*

The interconnections of FMU variables can lead to circular dependencies between the variables. An example of this behavior is the car suspension system that is presented in Sect. 4. Figure 1 shows the co-simulation scenario of the example and the Initialization Graph of the system. The Initialization Graph in Fig. 1 is annotated with the strongly connected components of the graph.

The following definitions formalize the concept of an algebraic loop in a co-simulation scenario and define the problem these algebraic loops are introducing. The definition of strongly connected components is adapted from the semantics of Causal Block Diagrams (see [8] for an overview).

Definition 9 (Algebraic loops). *An algebraic loop is defined as a non-trivial, strongly connected component of the graph in Definition 8. Formally, a strong connected component satisfies $\{a, b \in SCC : Path(a, b)\}$, where $Path(a, b)$ is true when there's a path (including an empty path from a node to itself) between nodes a and b ($Path(a, a)$ is always true). An SCC is non-trivial when it has more than one node.*

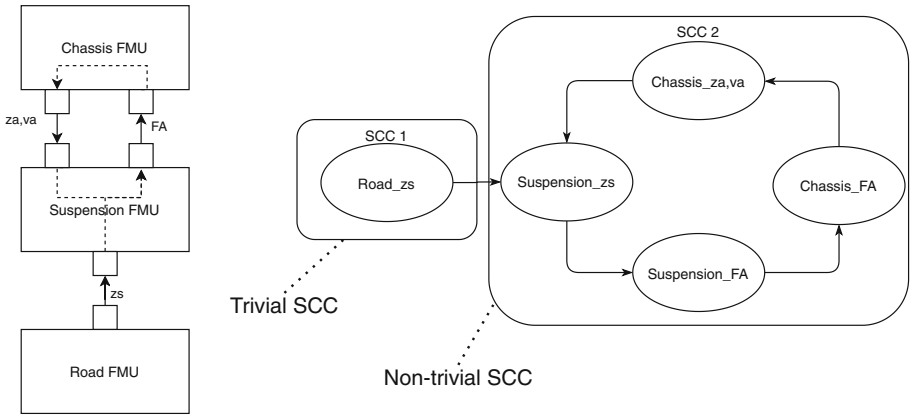

Fig. 1. An FMU co-simulation scenario of the Quarter car and its Initialization Graph denoted with SCCs.

Since the edges of the graph represent dependencies between the variables, the value of every variable in a non-trivial strong component depends on itself. Let X denote a vector of one or more variables whose value depends on itself. The non-trivial strong component forms an equation with the form $F(X, U) = X$, where F denotes the relations between the variables in the loop and U denotes the variables whose values are calculated elsewhere. This means that algebraic loops need to be handled using fixed point iterations[12].

An example of a co-simulation scenario where fixed point iteration is needed can be seen in Fig. 1 where the Initialization Graph of the quarter car system from Sect. 4 is shown.

A fixed point iteration technique is not guaranteed to convergence if the system is unstable. The fixed point is as a numerical fixed point that approximates a limit if such a value exist (the system is stable). It means that an upper bound of the number of repetitions needs to be established to ensure termination. In the case of a non-converging algebraic loop, the simulation should be stopped since the result of the co-simulation scenario would not be trustworthy. The criteria of a valid co-simulation scenario are specified in Definition 10.

Definition 10 (Convergence of Fixed point iteration). *A fixed point iteration converges if a finite number of iterations will make the difference of the output value of the same operation between two following iterations within a certain threshold* ϵ.
Formally, $\exists n \in \mathbb{N} : |F(X^{n+1}, U) - F(X^n, U)| \leq \epsilon$.

2.2 INTO-CPS Maestro 2

INTO-CPS Maestro 2[1] [22] is a framework for creating simulation specifications and executing such specifications. The framework is FMI-based and set to supersede Maestro [21] with the main goal of supporting research into co-simulation based on FMI.

The philosophy of the framework is to separate the specification of a co-simulation from the execution. This allows one to inspect and verify, manually or automatically, how a given co-simulation is to be executed.

A specification is expressed in the domain-specific language called Maestro Base Language (MaBL), and it is explicit, such that the application of, i.e., FMUs are transparent. Expansion plugins can assist in creating such MaBL specifications, and one can apply expansion plugins that, in turn, generate the MaBL code. The plugin described in this paper is such an expansion plugin. The application of a plugin is evident in a MaBL specification. Upon processing of the specification, a new specification is created where the application of a plugin is replaced by the MaBL code generated by the plugin. This process is known as expansion, and a specification without any expansions remaining is a fully expanded MaBL specification. An example of a part of the folded MaBL specification of the case study example of Sect. 4 can be seen below.

```
simulation
import Initializer;
{
FMI2 chassis = load("FMI2", "{8c4e810f-3df3-4a00-8276-176
    fa3c9f000}", "src/chassis-c.fmu");
...
IFmuComponent components[3]={chassis,suspension, road};
expand initialize(components,START_TIME, END_TIME);
...
}
```

To conduct a co-simulation, Maestro2 also features an interpreter that can execute a fully expanded MaBL specification, resulting in the execution of the co-simulation.

3 Calculation of an Initialization Order

The FMI specification defines certain information about the initialization order described through different states of a co-simulation. The initialization phase covers the two states (in chronological order) defined in the FMI specification:

- *Instantiated*
- *Initialization Mode*

[1] Currently in alpha https://github.com/INTO-CPS-Association/maestro/tree/2.0.0-alpha.

In each of the two states, different groups of FMU variables and parameters are potentially assigned a value. The groups are defined by FMI based on the characteristics of the FMU variables. The rules have been extracted as predicates and used in the implementation. Some groups consist of variables and parameters whose value does not depend on other variables. These independent variables and parameters can be set in the *Instantiated* phase of the Initialization. Since these variables have no connections to other FMU variables - meaning they are not represented in the graph of Definition 8, the order their value is set in is insignificant. The setting and getting operations of each FMU are grouped to perform the fewest possible FMU-operations during the Initialization.

In the *Initialization Mode* state all the interconnected variable is being defined, but as stated by the Definitions 3, 4 and 7 the operations *get* and *set* **require** that the operations are performed in a specific order. Furthermore, algebraic loops place even more requirements on the initialization strategy. Since each non-trivial strongly connected component (algebraic loop) needs to be isolated from the other variables of the system to calculate their initial values using fixed point iteration as described in Definition 5. After the *Initialization Mode* state, all variables of all FMUs in the co-simulation scenario should be defined, and the co-simulation should be ready to enter the *modelInitialized* state.

3.1 Method to Calculate the Initialization Order

This section describes the approach taken to calculate the initialization order of the interconnected FMU variables. The approach is based on the strategy proposed in Gomes et al. [5,10], but the approach in this work is extended with the ability to handle the Initialization of algebraic loops.

The initialization algorithm starts by building a directed graph of the dependencies between the interconnected variables of the FMUs. The graph is constructed based on the interconnected variables and internal connections (feedthrough); please see Definition 8 for a formal definition of the graph.

The topological ordering of the strongly connected components of the graph defined in Definition 8 is the initialization order of the interconnected FMU variables. The non-trivial strongly connected components are algebraic loops of the system. The trivial ones are standard interconnected FMU variables, whose port operation should be performed only once during the initialization procedure. The calculation of an initialization order is performed in linear time based on the number of external and internal connections using Tarjan's algorithm [20].

As described in earlier sections, it is essential to handle algebraic loops by a particular initialization strategy since the loops otherwise would invalidate the co-simulation result. The procedure for initializing algebraic loops is identifying and initializing them using a fixed point iteration strategy until convergence. Since convergence is not guaranteed, this property is monitored using Definition 10 to

see if the difference between all the output variables of two successive iterations is below a defined threshold. Suppose convergence is not established within a finite number of iterations[2], the co-simulation scenario is rejected to avoid running an invalid simulation.

3.2 Optimization of a Initialization Procedure

An initialization procedure can sometimes be optimized since the FMI specification allows multiple *set* or *get* operations of the same FMU to be performed in bulk by grouping them together to a single operation over multiple variables with similar characteristic. This criteria of optimization is formalized in Definition 11.

Definition 11 (Optimization of a Initialization procedure). *Given an initialization procedure* $(I_i)_{i \in \mathbb{N}}$ *with a finite ordered sequence of FMU function calls* $f_i \in F = \bigcup_{c \in C} \{\texttt{set}_c, \texttt{get}_c\}$, *and* i *denoting the order of the function call. It can be optimized if* $\exists f_i, f_{i+1} \in F : \exists c \in C : (f_i \in \texttt{set}_c \wedge f_{i+1} \in \texttt{set}_c) \vee (f_i \in \texttt{get}_c \wedge f_{i+1} \in \texttt{get}_c)$

The correctness of the optimization in Definition 11 is established by the proof of using the Initialization Graph's topological ordering as the initialization order by Gomes et al. [11]. Their proof is trivially shown to cover this approach since the optimization does not change the structure of the Initialization Graph. A limitation of this optimization strategy is that it is not guaranteed to find all potentially valid optimizations of a co-simulation scenario. Considering it works only on a specific co-simulation step (a topological order of a graph), which is not necessarily unique for a given co-simulation scenario. A more advanced optimization strategy needs to be developed to perform all viable optimizations of a co-simulation step. Another solution is to apply this optimization strategy on the set of all valid co-simulation steps - yielding a potential very inefficient initialization algorithm. The initialization of a co-simulation is typically not the most time consuming or computational heaviest part of the co-simulation. However, it is still considered a low hanging fruit to apply this optimization to optimize the initialization.

3.3 The Complete Initialization Strategy

The pseudo-code in Algorithm 1 formulates the entire initialization strategy of the interconnected variables of a co-simulation scenario.

[2] 5 iterations is the default in our approach. This number is based on experience.

Algorithm 1. Initialization strategy for Interconnected variables

```
1: InitializationGraph ← createGraph(connections)
2: SCCS ← Tarjan(InitializationGraph)
3: OptimizeInitializationOrder(SCCS)
4: for each: SCC ∈ SCCS do
5:     if isAlgebraicLoop(SCC) then
6:         applyFixedPointIteration(SCC);
7:     else
8:         initializeVariable(SCC);
9:     end if
10: end for
```

As seen from the algorithm in Algorithm 1, the algebraic loops are handled using a different initialization strategy compared to the other trivial SCC of a single interconnected FMU variable.

4 Case Study

In this section, we give a simple example of a co-simulation whose correct initialization demands the solution to an algebraic loop.

We consider a co-simulation of a quarter car model [19, Section 6.4], illustrated in Fig. 2. We omit the equations that each FMU is solving but note that gravity acts on both wheel and chassis masses and that the origin of each mass is when the springs are not displaced. The equations and simulation model for this example are available online[3].

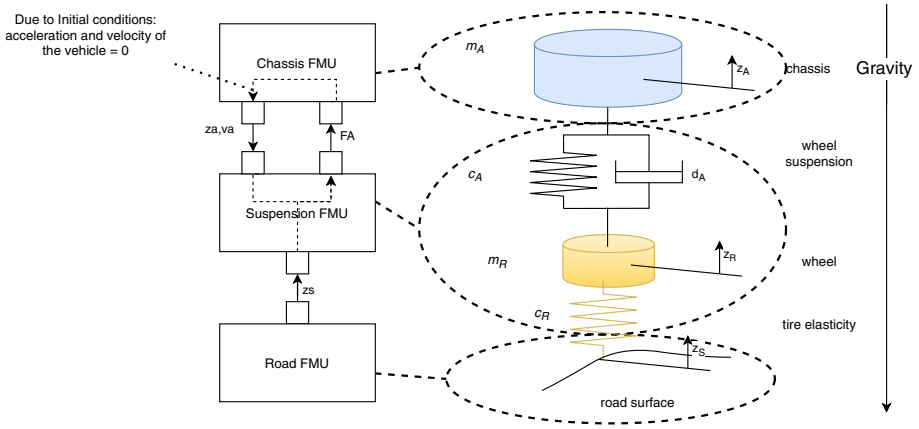

Fig. 2. Quarter car model co-simulation. Adapted from [19, Section 6.4].

[3] https://github.com/SimplisticCode/QuarterCarCaseStudy.

The FMUs need initial conditions specified by equations that restrict the possible initial values for the position and velocity of the wheel and chassis masses. Figure 3 illustrates what happens when we set those positions and velocities to zero. Note that, because of gravity, the car chassis bounces on the suspension wheel, with a maximum compression of about 17cm compared to when the system's springs are uncompressed. This is most likely an invalid scenario, as the car's suspension might not be rated to be displaced that much. In any case, the purpose of simulation studies involving quarter car models is to understand how well a suspension system absorbs shock when the car goes over a bump, not when the car *falls on the road*, which is what the simulation results in Fig. 3 resemble.

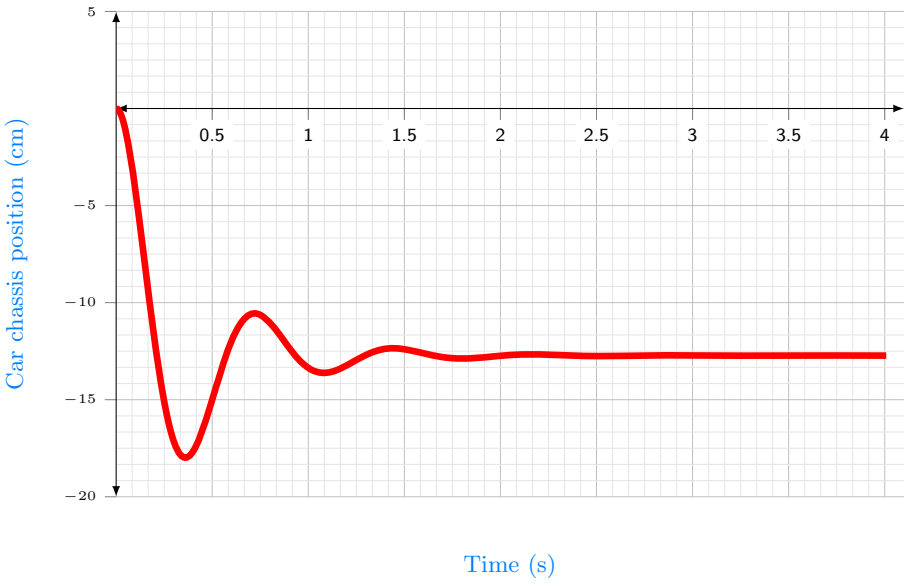

Fig. 3. Simulation results when position and velocity of the chassis mass is zero.

The correct way to initialize this co-simulation scenario is to force the master algorithm to calculate the valid initial velocities and position from equations that force the accelerations and velocities on the masses to be zero. This will force the co-simulation to initialize to a steady state.

To make the above explanation concrete, we now show the equations that are active at the initial time for each FMU for a correct initialization, and we show that there is an algebraic loop.

For the road FMU, the initial equation is simply the initial height of the road surface, which in this case is zero, i.e., $z_s = 0$. For the suspension FMU, the following equations are active:

$$a_R = 0.0 \quad \text{Acceleration of tire} \quad (1)$$
$$v_R = 0.0 \quad \text{Velocity of tire} \quad (2)$$
$$F_{gR} = 9.81 * m_R \quad \text{Gravity on the tire} \quad (3)$$
$$F_R = -c_R * z_R \quad \text{Rubber force acting on tire} \quad (4)$$
$$F_A = c_A * (z_A - z_R) + d_A * (v_A - v_r) \quad \text{Suspension force acting on tire} \quad (5)$$
$$F_{total} = F_R + F_A - F_{gR} \quad \text{Total forces acting on tire} \quad (6)$$
$$a_R = (1/m_R) * F_{total} \quad \text{Acceleration of tire.} \quad (7)$$

Finally, for the Chassis FMU, the following equations are active at the initial time:

$$a_A = 0.0 \quad \text{Acceleration of chassis} \quad (8)$$
$$v_A = 0.0 \quad \text{Velocity of chassis} \quad (9)$$
$$F_{gA} = 9.81 * m_A \quad \text{Gravity on the chassis} \quad (10)$$
$$a_A = (1/m_A) * (-F_A - F_{gA}) \quad \text{Acceleration of chassis.} \quad (11)$$

To see that there is an algebraic loop, note that the output z_A of the chassis FMU is not restricted directly, but instead has to be computed from the acceleration equations $a_A = 0 = (1/m_A) * (-F_A - F_{gA})$. The later contains the output F_A of the Suspension FMU. This output, in turn, depends on z_A, thus yielding

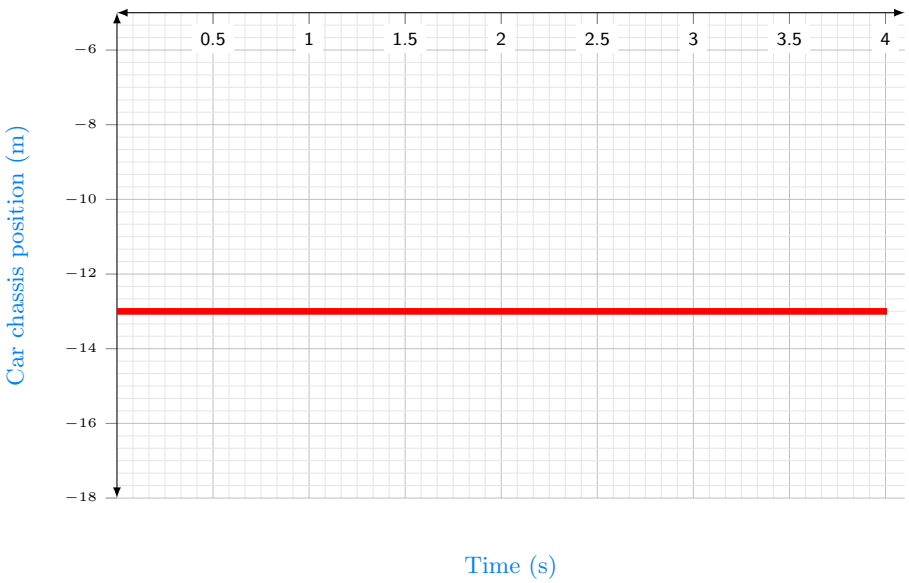

Fig. 4. Simulation results starting from a correct initial state (a steady state).

an algebraic loop. Figure 4 shows the simulation results when the algebraic loop is properly solved during initialization.

5 Realization of a Maestro 2 Plugin

The presented approach has been realized as a Maestro 2 expansion plugin that generates the *Initialization*-phase of a co-simulation specification expressed in MaBL. The plugin calculates the MaBL-specification based on the FMUs of a co-simulation scenario and a specific plugin-configuration to let the user supply the initial values of FMU parameters and fine-tune the initialization of the system. The plugin can calculate a correct initialization specification if the co-simulation scenario adheres to the behavior dictated by the definition given in Definition 10 meaning all algebraic loops in the scenario convergences within a finite number of iterations.

The plugin optimizes the initialization order by grouping operations that can be executed in *parallel* to take advantage of FMI's ability to *set* or *get* multiple variables of a single FMU in bulk. The criteria for this optimization is defined in Definition 11. The developed plugin has been tested on numerous co-simulation scenarios from the INTO-CPS universe[21] and compared with the existing *Initializer* of Maestro. The plugin has been tested as a part of the complete Maestro 2 pipeline.

5.1 Realization of the Topological Sorting

The topological sorting algorithm (Tarjan's Algorithm) is implemented in Scala [18], an object-oriented programming language incorporating many features from the functional programming paradigm. The motivation for choosing Scala [18] is its relation to JVM and the connection to Slang and the Sireum framework [17]. Slang (Sireum Language) is a programming language based on Scala, developed at Kansas State University (KSU), to develop and reason about critical software systems. Sireum is a framework for performing programming language analysis, reasoning, and verification of CPS also developed at KSU. Logika is one of the tools in the Sireum framework used for performing automated formal verification of a piece of Slang code using the theorem prover Z3 [23]. The connection of the implementation to Slang and Logika will be investigated in future work. The plan is to use the Logika framework to formally verifying the plugin. This will also be used to explore how Slang's contract-based nature can be used to obtain more reliable results of co-simulations. Tarjan's algorithm returns a topological order of strongly connected components. The returned order is the initialization order, where the non-trivial strongly connected components denote an algebraic loop requiring a particular initialization strategy.

5.2 Verification of the Initialization Order

The plugin is verified using several methods. The plugin approach is established using traditional proof methods, and the plugin has been practically verified

against an established co-simulation step verifier. Gomes et al. have verified the approach in [9]; they proved the correctness of using the topological order of a dependency graph of the interconnected FMU-variables as the order of the operations in a co-simulation step (both the initialization procedure and an arbitrary step). Gomes et al. [9] used a graph of FMU-operations (*Set, Get, doStep*) in their proof instead of interconnected FMU-variables, which is the approach of this paper. The simplification of using the interconnected FMU-variables is valid and preserves the properties proved by Gomes et al. since this approach only considers the initialization phase of a co-simulation. This makes it possible to omit all the *doStep* nodes from Gomes et al.'s graph, eventually ending up with a graph similar to the initialization graph described in Definition 8. This approach is a subgraph of the graph by Gomes et al.[9], which allows their proof to be modified to the approach presented in this paper.

Practical Verification Against an Established Verifier. Gomes et al.'s [9] main contribution is a Prolog implementation of the principles for a valid FMI based co-simulation step [4]. Gomes et al. use the Prolog implementation in their research to verify their approach for generating different co-simulation algorithms. The Prolog realization encapsulates all the rules of a valid co-simulation step (both master-algorithm and an initialization algorithm). The Initializer includes an integration to the Prolog Verifier. The integration is a Java program based on JIProlog [13] - a library that allows calling Prolog predicates directly from Java. The integration is used to check the initialization order against the rules in the Prolog database. The integration performs all the necessary transformations of the dependency graph (see definition 8) used in the Maestro plugin to a graph of FMU operations used in the Prolog database. The transformation is based on the definitions 3 and 4. The integration has been realized to systematically verify the calculated initialization order's correctness against an established and recognized co-simulation Algorithm Verifier. The Prolog implementation does support co-simulation scenarios containing algebraic loop, so these scenarios are not tested against the Prolog database.

6 Related Work

Prior work [5,11] is looking into the generation of co-simulation algorithms (both master and initialization algorithms) for FMI-based scenarios. Their generation technique is like ours, based on a dependency graph of the operations of a co-simulation step. Both Gomes and Broman present an approach for using the topological order of a dependency graph to establish a correct order of operations in a co-simulation step of a given co-simulation scenario. The work by Gomes et al. [11] does also define the criteria for a correct co-simulation step. Their work has many similarities with ours. However, their work is mostly concerned with the theoretical aspect of co-simulation algorithm generation and

[4] http://msdl.cs.mcgill.ca/people/claudio/projs/PrologCosimGeneration.zip.

verification, while our work has a more practical nature. Gomes et al. do also not consider the handling of algebraic loops, which is a key feature of our approach. Furthermore, the approach taken in our work is only concerned with the initialization procedure of a co-simulation.

Broman et al. [5] also suggest to use the topological sorting of a dependency graph of the interconnected variables to detect algebraic loops and discover the partial order of port-operations. Nevertheless, they explicitly specify the requirement for cycle freedom in the dependency graph as a precondition for generating a valid co-simulation. It means they refuse all co-simulation scenarios containing algebraic loops. It is a significant difference to our approach that applies a fixed point iteration strategy to handle these scenarios. Also, the approach in this paper is more specialized because it only considers the initialization of a co-simulation, which means it deals with non-interconnected variables.

Amalio et al. [1] investigate how to avoid algebraic loops in FMU based co-simulation scenarios by statically checking the architectural design of a CPS. The publication's purpose is like ours, to avoid invalid co-simulation scenarios. Nevertheless, they achieve this by excluding co-simulation scenarios containing algebraic loops. Their method is realized in a co-simulation tool, INTO-SysML [15]. Formal methods form the basis of their work (Theorem Proving and Model-checking). It will be an inspiration for the future work of formally verifying the plugin and other parts of Maestro 2.

The work by Gomez et al. [6] is similar to ours. They use Tarjan's SCC algorithm to generate a sorted DAG of strongly connected components to solve the initialization problem. Even though their work is very similar to ours, we extend their approach with the verification against the simulation semantics resulting in a formally more sound approach. However, further work will look into further improvements and formal verification of the current approach.

7 Concluding Remarks

This work uses a topological ordering of a dependency graph of the interconnected FMUs variable and internal FMU connections along with predicates from the FMI specification to calculate a correct initialization order for a co-simulation scenario potentially containing algebraic loops. The initialization procedure optimizes the initialization order by grouping variables with similar characteristics to perform the fewest possible operations in the initialization procedure. This approach supports the initialization of a co-simulation scenario containing algebraic loops by using fixed point iteration. The approach is suitable to combine with well-established master algorithms like Gauss-Seidel and Jacobi [16]. The approach is realized as an expansion plugin for the open-source INTO-CPS Maestro 2 tool and verified against the existing *Initializer* and the calculated initialization order was verified against an established co-simulation Algorithm Generator and Verifier implemented in Prolog [9].

Future work includes formal verification of the plugin using the Logika framework[17]. We will also look into the generation of a verification strategy for

the whole Maestro 2 framework to examine how different forms of verification jointly can extend the trust of the correctness of the result of a co-simulation.

Acknowledgements. We would like to thank Stefan Hallerstede, Christian Møldrup Legaard, and Peter Gorm Larsen for providing valuable input to this paper and the developed plugin.

References

1. Amálio, N., Payne, R., Cavalcanti, A., Woodcock, J.: Checking SysML models for co-simulation. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 450–465. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_28
2. Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation v2.0. In: Schöps, S., Bartel, A., Günther, M., ter Maten, E.J.W., Müller, P.C. (eds.) Progress in Differential-Algebraic Equations. DEF, pp. 107–125. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44926-4_6
3. Bastian, J., Clauß, C., Wolf, S., Schneider, P.: Master for co-simulation using FMI. In: 8th International Modelica Conference, pp. 115–120. Linköping University Electronic Press, Linköpings universitet (2011). https://doi.org/10.3384/ecp11063115
4. Blockwitz, T., et al.: Functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In: 9th International Modelica Conference, pp. 173–184. Linköping University Electronic Press (2012). https://doi.org/10.3384/ecp12076173
5. Broman, D., et al.: Composition of FMUs for Co-Simulation. Technical report, University of California, Berkeley (2013). http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-153.html
6. Évora Gómez, J., Hernández Cabrera, J.J., Tavella, J.P., Vialle, S., Kremers, E., Frayssinet, L.: Daccosim NG: co-simulation made simpler and faster. In: The 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019, pp. 785–794, February 2019. https://doi.org/10.3384/ecp19157785
7. FMI.: Functional Mock-up Interface for Model Exchange and Co-Simulation (2014). https://fmi-standard.org/downloads/
8. Gomes, C., Denil, J., Vangheluwe, H.: Causal-block diagrams: a family of languages for causal modelling of cyber-physical systems. Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems, pp. 97–125. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43946-0_4
9. Gomes, C., Lucio, L., Vangheluwe, H.: Semantics of co-simulation algorithms with simulator contracts. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (2019). https://doi.org/10.1109/models-c.2019.00124
10. Gomes, C., Thule, C., Lúcio, L., Vangheluwe, H., Larsen, P.G.: Generation of co-simulation algorithms subject to simulator contracts. In: Camara, J., Steffen, M. (eds.) SEFM 2019. LNCS, vol. 12226, pp. 34–49. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57506-9_4
11. Gomes, C., et al.: HintCO - hint-based configuration of co-simulations. In: International Conference on Simulation and Modeling Methodologies, Technologies and Applications, pp. 57–68 (2019). https://doi.org/10.5220/0007830000570068

12. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a survey. ACM Comput. Surv. **51**(3), 1–33, Article 49 (2018). https://doi.org/10.1145/3179993
13. JIProlog: JIProlog, October 2016. http://www.jiprolog.com. Accessed 20 Aug 2020
14. Kübler, R., Schiehlen, W.: Two methods of simulator coupling. Math. Comput. Model. Dyn. Syst. **6**(2), 93–113 (2000). https://doi.org/10.1076/1387-3954(200006)6:2;1-M;FT093
15. Miyazawa, U.Y.A., Woodcock, U.J.: Integrated tool chain for model-based design of CPSs foundations of the SysML profile for CPS modelling (2016). https://www.semanticscholar.org/paper/INtegrated-TOol-chain-for-model-based-design-of-of-Miyazawa-Woodcock/3042572251aba18ab21ced9cc2fb49223dea2a2c. Accessed 13 Nov 2020
16. Palensky, P., Van Der Meer, A.A., Lopez, C.D., Joseph, A., Pan, K.: Cosimulation of intelligent power systems: fundamentals, software architecture, numerics, and coupling. IEEE Ind. Electron. Mag. **11**(1), 34–50 (2017). https://doi.org/10.1109/MIE.2016.2639825
17. Robby Hatcliff, J., Belt, J.: Model-based development for high-assurance embedded systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Modeling, pp. 539–545. Lecture Notes in Computer Science, Springer International Publishing (2018). https://doi.org/10.1007/978-3-030-03418-4_32
18. Scala: The Scala Programming Language, August 2020. https://www.scala-lang.org. Accessed 19 Aug 2020
19. Schramm, D., Hiller, M., Bardini, R.: Force components. Vehicle Dynamics, pp. 207–224. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-54483-9_9
20. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010
21. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.: Maestro: the INTO-CPS co-simulation framework. Simul. Model. Pract. Theor. **92**, 45–61 (2019). https://doi.org/10.1016/j.simpat.2018.12.005
22. Thule, C., et al.: Towards reuse of synchronization algorithms in co-simulation frameworks. In: Camara, J., Steffen, M. (eds.) SEFM 2019. LNCS, vol. 12226, pp. 50–66. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57506-9_5
23. Z3prover: z3, September 2020). https://github.com/Z3Prover/z3/wiki. Accessed 13 Sept 2020