Third party model integration in a modeling and simulation platform

Conquet, Janice C.R.

Award date:
2020

Department of Mathematics and Computer Science
Software Engineering and Technology Group

Third party model integration in a modeling and simulation platform

Master Thesis

Author:
J.C.R. Conquet
0877051
j.c.r.conquet@student.tue.nl

Supervisors:
M.G.J. van den Brand
B. Combemale

August 27, 2020

Abstract

Safran developed a modeling and simulation platform around geometrical and functional shared models. The various components of the models are designed by different designers, integrators, and architects. The idea is to allow the various models implemented in Python to be complemented by third party models like Modelica (Modelica Association, nd). Different techniques and concepts are explored to reach this goal.

The FMI standard is a tool independent standard that supports model exchange and co-simulation of dynamic models (FMI, nd). Using this standard we are able to extend the platform with third party models. Questions whether there are operations we can reuse for the connections between the various models is also looked at. These operations are located and used to generate the Python file is. A JSON file is used as input when generating the Python file. To get this JSON file, a simple Domain Specific Language is developed where models in the platform can be declared. Future improvements are to add logic and semantics into the JSON file, add a visual DSL instead of a written one, and experiment more with the master algorithm. The master algorithm is the algorithm that orchestrates the communication between the different models (Van Acker et al., 2015).

Preface

For my master thesis I was given the opportunity to work on a project at Inria in collaboration with Safran. I would like to thank Inria for giving me this opportunity. I would specifically like to thank my supervisor Benoit Combemale, as well as Frédéric Collonval and Étienne Lac from Safran for the guidance and support they gave me during this project. I also want to thank the Diverse team at Inria, which made me feel welcome and part of the team during my time at Inria. Furthermore, I would like to give special thanks to Eindhoven University of Technology. Specifically I want to thank my supervisor Mark van den Brand for introducing me to this interesting project, and guiding me through its process. Lastly, I want to thank my family and friends for the support they have given me during my master thesis.

Contents

1 Introduction **5**
- 1.1 Background . . . 5
  - 1.1.1 Co-simulation and Hybrid Co-simulation . . . 5
  - 1.1.2 FMI and FMU . . . 6
  - 1.1.3 Master Algorithm . . . 7
- 1.2 Objective . . . 8

2 Related Work **9**
- 2.1 Literature Study . . . 9
  - 2.1.1 Application of the Strategies . . . 9
  - 2.1.2 Result . . . 11
- 2.2 Discussion Literature Study . . . 13
  - 2.2.1 FMI extension for time step size . . . 13
  - 2.2.2 Wrapper for FMI . . . 13
  - 2.2.3 Semantic Adaptation . . . 14
  - 2.2.4 Decentralized data exchange FMU . . . 15
  - 2.2.5 Conclusion . . . 15
- 2.3 Additional Related Work on Master Algorithm . . . 15
- 2.4 Literature Study Conclusion . . . 17

3 Background Modeling and Simulation Platform **18**
- 3.1 Models within the Modeling and Simulation Platform . . . 18
- 3.2 Simulation within Modeling and Simulation Platform . . . 19
- 3.3 Discussion of Modeling and Simulation Platform . . . 20

4 FMU simulation in Python **21**
- 4.1 Simulation with FMPy . . . 21
- 4.2 Different step sizes . . . 25

5 Adjusting Modeling and Simulation Platform **27**
- 5.1 Integration FMU within Modeling Platform . . . 27
- 5.2 Generation of Python code using JSON . . . 30
- 5.3 DSL to generate JSON . . . 35
  - 5.3.1 Syntax . . . 35
  - 5.3.2 Static Semantic . . . 38
  - 5.3.3 DSL . . . 41
  - 5.3.4 Discussion . . . 43
- 5.4 Master Algorithm . . . 44
  - 5.4.1 Master Algorithm within Modeling and Simulation Platform 44
  - 5.4.2 Future improvements Master Algorithm . . . 44
  - 5.4.3 Conclusion . . . 45

6 Discussion **46**
6.1 Objective 46
6.2 Research Questions 47

7 Conclusion and Future Work **48**

References **49**

1 Introduction

Safran is an international high-technology group, operating in the aviation (propulsion, equipment and interiors), defense and space markets (Safran, nd). They are often modeling Cyber-Physical System. Cyber-Physical System is the integration of computation and physical processes (Lee, 2008). For this purpose, they have developed an inhouse modeling and simulation platform.

This is a platform around geometrical and functional shared models. It allows simultaneous work on the same model. This implies that different components of a model are developed by different designers, integrators, and/or architects. The core is developed in Python and user interactions are done through a Jupyter Notebook. The main objective of this project is to allow the platform users to extend the various models implemented in Python with possibly third-party models, for example Modelica (Modelica Association, nd), and to support the co-simulation of various heterogeneous models. Co-simulation is the theory and techniques to enable global simulation of coupled system via composition of simulators (Gomes et al., 2017b). Heterogeneous models are models modeling different subsystems using various languages and tools (Tripakis, 2015).

The remainder of this section will provide a background on the various techniques and concepts used throughout this thesis. It also gives an overall view of the objective of this thesis and the problems Safran is facing with the current state of the platform.

1.1 Background

In this section we will provide the background of several techniques and concepts used throughout this project.

1.1.1 Co-simulation and Hybrid Co-simulation

As mentioned before, co-simulation is the theory and techniques to enable global simulation of a coupled system via composition of simulators. Each of these simulators is a black box mock-up of a subsystem within this system (Gomes et al., 2017b). There are three types of approaches to co-simulation, namely discrete event, continuous time, and hybrid co-simulation.

(Gomes et al., 2017b) describes the discrete event based co-simulation approach describes a family of orchestrators and characteristics of simulation units that are borrowed from the discrete event system simulation domain. The essential characteristics of a discrete event system are reactivity and transiency. Reactivity is the instant reaction to external stimuli by an external, and transiency is the system can change its state multiple times in the same simulated time point and receive simultaneous stimuli. A discrete event simulation unit is a

black box. It is also typical to assume that discrete event simulation units communicate with the environment via time-stamped events, as opposed to signals. Thus, the outputs of simulation units can be absent at times where no event is produced.

(Gomes et al., 2017b) describes the continuous time based co-simulation approach, the orchestrators' and simulation units'behavior and assumptions are borrowed from the continuous time system simulation domain. A continuous time simulation unit is assumed to have a state that evolves continuously over time. Each simulation unit's micro-step sizes are independent in continuous time co-simulation. To overcome this, a communication step size H, also known as macro-step size or communication grid size, has to be defined. H marks the times at which the simulation units exchange values of inputs/outputs.

(Gomes et al., 2017b) describes the hybrid co-simulation approach as mixing the characteristics and assumptions of discrete event and continuous time approaches. In order to understand this better, they give an example of a thermostat. The temperature in the room is being monitored by the sensor which is part of a continuous time system. The heater is either turned on or off and thus is part of a discrete event system. Together they make the hybrid system thermostat.

1.1.2 FMI and FMU

Another technique that is used is the Functional Mock-up Interface (FMI) standard (FMI, nd). The FMI standard is a tool independent standard to support both model exchange and co-simulation of dynamic models (FMI, nd). Functional Mock-up Unit (FMU) is a component which implements the FMI. It is a zipped file containing XML description file and the implementation in source or binary form (Blochwitz et al., 2011).

The FMI standard consists of two parts, namely FMI for model exchange and FMI for co-simulation. The difference between the two is that FMI for co-simulation implements a solver within the FMU, while FMI for model exchange does not.

The intention behind FMI for model exchange is that a modeling environment can generate C-code of a dynamic system model in the form of an input/output block (Blochwitz et al., 2011). This block can be utilized by other modeling and simulation environments.

The intention of FMI for co-simulation is to couple two or more simulation tools in a co-simulation environment (Blochwitz et al., 2011). The subsystems (slaves) are solved independently from each other by their individual solver, while a master controls the data exchange and synchronization between them. This master (master algorithm) is a solver algorithm. Section 1.1.3 will explain this in more detail. *Figure 2* shows the difference between FMI for model exchange and FMI for co-simulation.

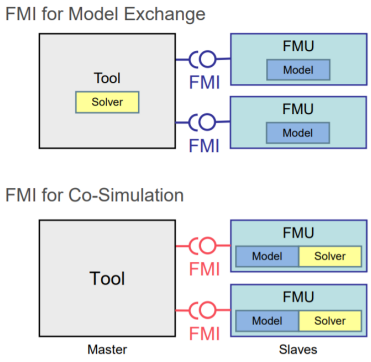


Figure 2: Difference between environment of FMI for model exchange and FMI for co-simulation (FMI, nd)

A component which implements the FMI is the FMU. Since FMUs represent dynamic models, they contain different variables. These variables in FMI for model exchange corresponds to the inputs and outputs of the input/output block. In FMI for co-simulation, they are used as input and output for the solver of each individual subsystem. To simulate the system as a whole over a period of time, time steps are used to exchange data of the variables of the subsystems with each other.

FMU consists of one zip file containing an XML file, a description file, and the implementation in source or binary form. The XML file contains the definition of all variables of the FMU that are exposed to the environment in which it will be used, as well as other model information (Blochwitz et al., 2011). For co-simulation all the slave specific information are contained in a slave specific XML file (Blochwitz et al., 2011). A small set of easy to use C functions are provided in source or binary form (Blochwitz et al., 2011). In the case of co-simulation, these will initiate a communication with a simulation tool to compute a communication time step, and to perform the data exchange (Blochwitz et al., 2011).

1.1.3 Master Algorithm

The master algorithm provides the orchestration for the entire co-simulation (*Figure 3*) (Van Acker et al., 2015). It orchestrates both data exchange and execution of the different FMUs in one co-simulation model. However, this master algorithm is not part of the standard (Van Acker et al., 2015) and thus the user needs to develop it themselves.
The master algorithm is part of the master slave structure of the FMI for co-simulation. The master algorithm controls the data exchange between the different FMUs, thus the FMUs are the slaves (solver) in this system.

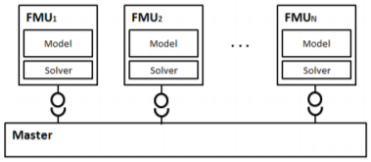


Figure 3: Co-simulation environment (Van Acker et al., 2015)

1.2 Objective

As mentioned before, Safran wants to allow third party models in their platform. The FMI standard will provide a good standard to allow these third party models into their platform. The platform will need to support importing FMUs and connect the existing models to these FMUs.

Furthermore, the connection between the different heterogeneous models seem quite straightforward. The inputs and outputs of the models need to be connected to each other. They want to know if there are components that regulate these connections (composition operators) that can be reused for different systems. Their specific question is in a context of grey box system. Grey box system is a system containing both white box models (all variables are accessible by other models) and black box models (only select variables are accessible by other models). Since the platform models are white box and the FMUs are black box, the system will be a grey box system.

Lastly, the platform currently requires basic knowledge of Python from the designers, integrators, and architects. Ideally they would want a platform that does not require this from its users. A Domain Specific Language (DSL) can be used to remove this knowledge barrier.

Based on these problems that Safran is currently facing, the research questions can be formulated as follows;

1. Can reusable composition operators in between heterogeneous models be defined, in a context of grey box co-simulation?

2. Can we reify the Python based abstractions into Domain Specific Languages to define such operators?

In the remainder of this thesis we will look at the related work, including a literature study on co-simulation and hybrid co-simulation (*Section 2*), followed by a background on Safran's modeling and simulation platform (*Section 3*). We will also look at simulating FMUs in Python (*Section 4*). We will then proceed with how we can adjust Safran's modeling and simulation platform, including integrating FMUs within the platform, generating the Python code using JSON, and how we developed a DSL to generate this JSON file (*Section 5*). Lastly, we will give a discussion, a conclusion and future work (*Section 6* and *Section 7*).

2 Related Work

Several papers have already addressed the challenges of hybrid co-simulation and the master algorithm. To give an overview of these paper, we will conduct a literature study surrounding co-simulation and hybrid co-simulation using FMI (*Section 2.1*). *Section 2.3* will expand on the literature study concerning the master algorithm.

2.1 Literature Study

We conducted a literature study on co-simulation and hybrid co-simulation using FMI. This literature study will bring us closer to answering the research questions in this thesis. Two strategies were used to find relevant papers for our questions, namely database searches and backwards snowballing strategies. Database searches involves starting with systematic searches in databases using well-defined search strings to find relevant literature (Jalali and Wohlin, 2012). Backwards snowballing means using the reference list to identify new papers to include (Wohlin, 2014).

To carry out the research in this section, a combination of the two strategies are used. A list of relevant papers were given initially, to which the backwards snowballing is applied. Keywords are thought of with the help of these papers and used in the database searches. Backwards snowballing is then applied on the papers found in this search.

2.1.1 Application of the Strategies

A list of literature was given, from which new papers were found using backwards snowballing. The most important paper from this list is the paper by (Gomes et al., 2017b). This paper is a survey conducted on co-simulation. It references different relevant papers. They are filtered by examining their title, followed by reading their abstract and conclusions. Five papers remain from this procedure.

Next to this technique, the database search technique was also used. The keywords "co-simulation", "hybrid co-simulation", and "hybrid co-simulation fmi" were used to find relevant literature on Google Scholar. The first 20 papers from each keyword is collected which amounts to 60 papers. They are filtered by reading the abstract and conclusions until 17 papers remained. These are thoroughly read and the 5 most relevant papers are chosen to apply backwards snowballing. This technique is applied in one degree due to the same set of papers being referenced. The workflow of this process can be seen in *Figure 4*. After removing duplicate papers, a total of 20 papers remained. These papers can be seen in *Table 1*. The papers are divided into relevant, semi-relevant, and irrelevant papers. A description explaining why they are relevant or not is given.

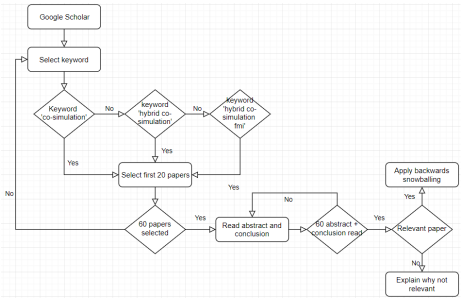


Figure 4: Workflow graph of the steps to get relevant papers.

Paper	*Description*
	Relevant
(Gomes et al., 2017b)	A survey explaining co-simulation. It references several different papers that use FMI for hybrid co-simulation.
(Bastian et al., 2011)	Addresses FMI for co-simulation.
(Broman et al., 2013)	Proposes FMI extension for predictable step sizes.
(Cremona et al., 2019)	Proposes FMI extension that includes integer time, automatic choice of time resolution, and the use of absent signals.
(Camus et al., 2016)	Uses a wrapper for hybrid co-simulations that relies on DEV&DESS. This is a black box.
(Denil et al., 2015)	Semantic adaptation using FMI. Four different ways, namely adaptation in master, wrapping the embedded model, adaptation in separate FMUs, adaptation on the encapsulating model.
(Tavella et al., 2016)	Proposes FMI extension to predict the time and step size.
(Cremona et al., 2016b)	Introduces an IDE to generate a pure C co-simulation implementation that is potentially very efficient and, like an FMU, can run on a wide variety of platforms. Proposes FMI extension to accept no signal (step size can be 0).
(Galtier et al., 2015)	Decentralizes data exchange between FMUs. a local master collects the proposed step size from each FMU on the same node and sends the local minimum value to the global master.
(Lee, 2008)	Explains Cyber-Physical Systems.
(Tripakis, 2015)	Proposes a timed wrapper for discrete, and proposes restrictions to handle non-deterministic state machines. This is to model them as FMUs.
(Sun et al., 2011)	Proposes FMI extension for step sizes.
	Semi-relevant
(Wetter, 2011)	Uses a middleware as opposed to linking clients directly. FMI is not used.

(Gomes et al., 2017a)	Uses a step size to do a stability analysis to ensure the stability properties of the original system.
(Cremona et al., 2016a)	Uses step revision. It offers a better insight on obtaining a step size with FMI.
(Bogomolov et al., 2015)	Explains how co-simulation between two heterogeneous systems, Uppaal and SpaceEx, works using FMI components.
(Feldman et al., 2014)	Proposes a FMI export fuctionality for Rhapsody.
Irrelevant	
(Lajolo et al., 1999)	Explains how to improve a compiled hardware/software co-simulation. This is done in C and does not use FMI standard.
(Widl et al., 2015)	Explains two approaches to closed-loop control system co-simulation.
(Van Acker et al., 2015)	Implements a master algorithm for the co-simulation.

Table 1: Papers divided by relevance and category.

2.1.2 Result

The total number of papers consulted per year and the total number of relevant papers per year are given in *Figure 5*. Here it can be seen that this topic is relatively new. Most papers are published in the last decade, with an increase of publications after 2015. 2015 has the highest number of publications. The topic of hybrid co-simulation, specifically with FMI and FMU, is still being researched as the latest paper was published in 2019. When looking at the relevant papers, the papers chosen are also from the period of 2015 to 2019.

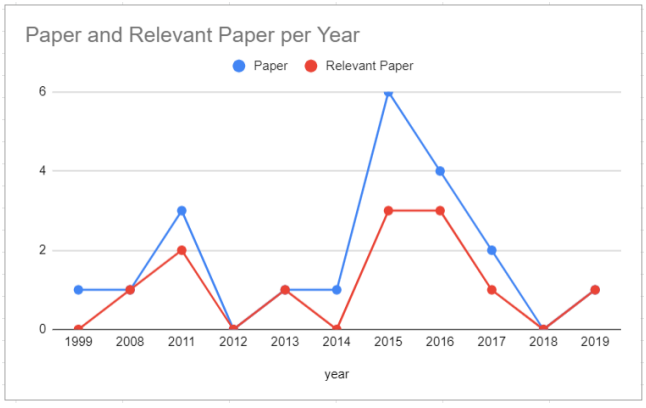


Figure 5: Papers and relevant papers per year.

The relevant papers can be divided into six categories as can be seen in *Figure 6*. The category with the most papers is the FMI extension with time step size. The second category with the most papers is wrapper for FMU.

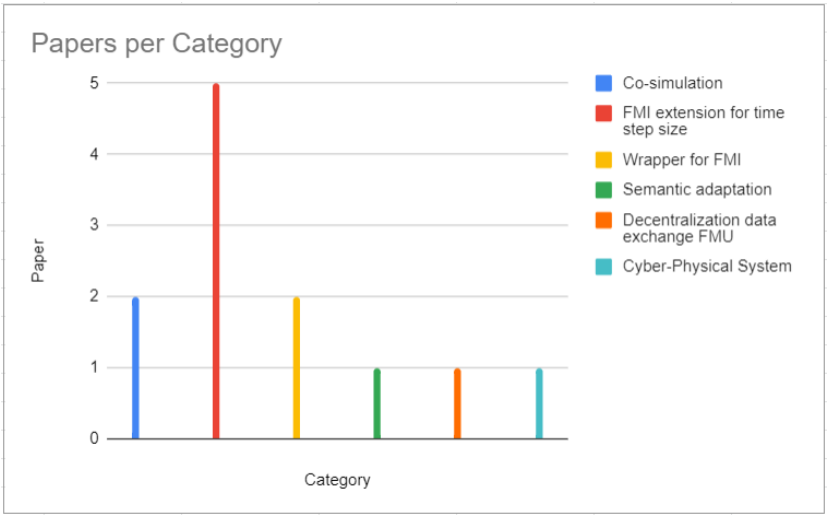


Figure 6: Relevant papers divided into categories.

From *Table 2* it can be seen that the FMI extension for step size has been researched several times. It is an ongoing topic to make the step size of the time being modelled in the FMI extension more efficient. The papers start in 2011 and continue till 2019. It is expected that this topic will have more publications in the future.

Category	Paper
Co-simulation	(Bastian et al., 2011) (Gomes et al., 2017b)
FMI extension for time step size	(Sun et al., 2011) (Broman et al., 2013) (Cremona et al., 2016b) (Tavella et al., 2016) (Cremona et al., 2019)
Wrapper for FMI	(Tripakis, 2015) (Camus et al., 2016)
Semantic adaptation	(Denil et al., 2015)
Decentralization data exchange FMU	(Galtier et al., 2015)
Cyber-Physical Systems	(Lee, 2008)

Table 2: Relevant paper per category.

2.2 Discussion Literature Study

(Gomes et al., 2017b) conducted a survey explaining co-simulation, including hybrid co-simulation. It references several other papers that also investigated the topic of hybrid co-simulation in the context of FMI. However, the main project is a generalization from a software engineering point of view. Thus, we have conducted a literature study of our own to investigates hybrid co-simulation in the context of FMI with this point of view.

From the literature study results there are five papers suggesting extending the FMI standard to handle discrete time co-simulation, two papers suggesting a wrapper for FMI, one paper suggesting semantic adaptation, and one paper suggesting decentralization data exchange.

2.2.1 FMI extension for time step size

(Sun et al., 2011) uses an inhouse FMU simulator which performs time integration with required time span and time step size. These time steps are restricted to an upper limit. (Broman et al., 2013) uses the master algorithm to determine a time step h. This time step is either accepted or rejected if it is too large. An FMU accepts time step h if the time passed until a state change is equal to the time step ($h' = h$). If it makes partial progress until ($h' < h$), then it must accept any time step h'' smaller than or equal to h', provided the FMU is started from the same original state. (Cremona et al., 2016b) uses this same method for the step size.

(Tavella et al., 2016) proposes to improve the time step size of the discrete event, or state event, also known as an unpredictable event. It introduces a function that allows the FMU solver to stop at the first unpredictable event with a new return code $fmi21Event$ and the time event is given by the returned value $nextEventTime$. If no event occurs, $nextEventTime$ is the computation step end time. So, the master algorithm can go on computing state variables without performing any rollback on this component.

(Cremona et al., 2019) extends the FMI by defining a semantic notion of time that can be used to model both continuous physical dynamics and discrete events. They use the same method as (Sun et al., 2011) and (Cremona et al., 2016b), but they use integer instead of float to represent time step. By using integer instead, the smallest representable time difference between two time stamps is always the same as the greatest common divisor and is thus always precise.

2.2.2 Wrapper for FMI

(Camus et al., 2016) suggest a wrapper for hybrid co-simulation of FMU components. The wrapper implements the simulation protocol functions for controlling the models evolution through the simulation software. Here the Discrete Event

System specification (DEVS) simulation protocol is used (Zeigler et al., 2018). The FMU uses the step size as described above to compute its next event internally. If the next internal event corresponds to a communication point of the FMU, then the wrapper retrieves the continuous output ports values, and produces the external output events accordingly. On the other hand, if the next internal event corresponds to a state-event or the next internal event of the discrete-event component, then the internal transition function of this latter is called, which could produce external output events.

(Tripakis, 2015) on the other hand uses a periodic wrapper to make untimed state machines timed. A periodic wrapper approach for encoding state machines as FMU can be seen as wrapping the machine with a periodic sampler at the inputs and a "hold" at the outputs. Inputs are sampled every T time units, while outputs remain constant until the next sampling occurs. They also mention encoding of discrete event, dataflow, and timed automata as FMU.

2.2.3 Semantic Adaptation

(Denil et al., 2015) suggest four different of semantic adaptation. The first one suggests that the master algorithm implements the state event detection and location algorithms as well as the data adaptations. The master algorithm is not reusable and complicated.

The second method suggests that the embedded model is wrapped completely by the semantic adaptation. The proposed approach results in a lot of communication overhead between the master and the different slaves. The wrapper of the embedded model is not generic and needs to be regenerated. However, the master algorithm is straightforward and reusable.

The third method suggest creating two additional FMUs, adhering to the FMI standard. The first FMU, the State Event Locator (SEL), detects and locates the events. The Transducer (TD) is responsible for translating events to signal values. The wrappers for both are reusable in all situations. The master algorithm is straightforward and reusable, and each FMU can be replaced with a different FMU or a real-world component.

The fourth method suggest that the encapsulating model is adapted with the State Event Location and Transducer. When an event is located without enough precision, the step is rejected by the slave. However, the slave FMU can use the location mechanism internally to compute the correct time step for the master and provide it when queried for the last successful time. This results in less communication on the bus, but is less straightforward to plug in different FMUs or real-world components.

2.2.4 Decentralized data exchange FMU

(Galtier et al., 2015) proposes to decentralize the data exchange between FMUs. The master functionality is carried out by a combination of three kinds of components distributed across the various computation nodes. *Figure 7* shows the architecture. The FMU wrapper handles direct interactions with other FMUs and data exchange with other FMU wrappers. All the FMUs perform a simulation step, which is the same for all FMUs, with a simple co-simulation schema. After the simulation of step i, each FMU examines its outputs and estimates how far they are from the exact value.

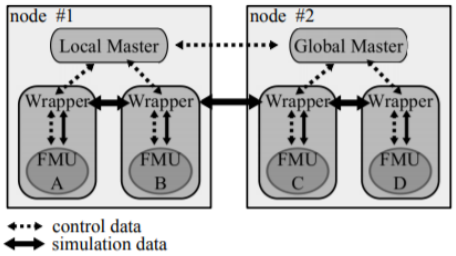


Figure 7: Co-simulation components architecture (Galtier et al., 2015).

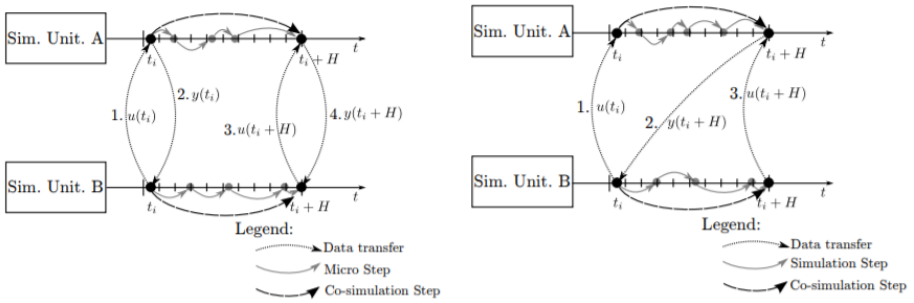
2.2.5 Conclusion

The literature study focused on co-simulation and hybrid co-simulation. From the results we can see that there are several suggestions on how to tackle the problem of co-simulation. However, from all these suggestions, it can be seen that the communication and the communication step between the different models plays a big role in co-simulation and hybrid co-simulation.

2.3 Additional Related Work on Master Algorithm

The communication and the communication step between the different models plays a big role in co-simulation and hybrid co-simulation. These are regulated and maintained by the master algorithm. Thus, when talking about them, we also have to talk about the master algorithm.

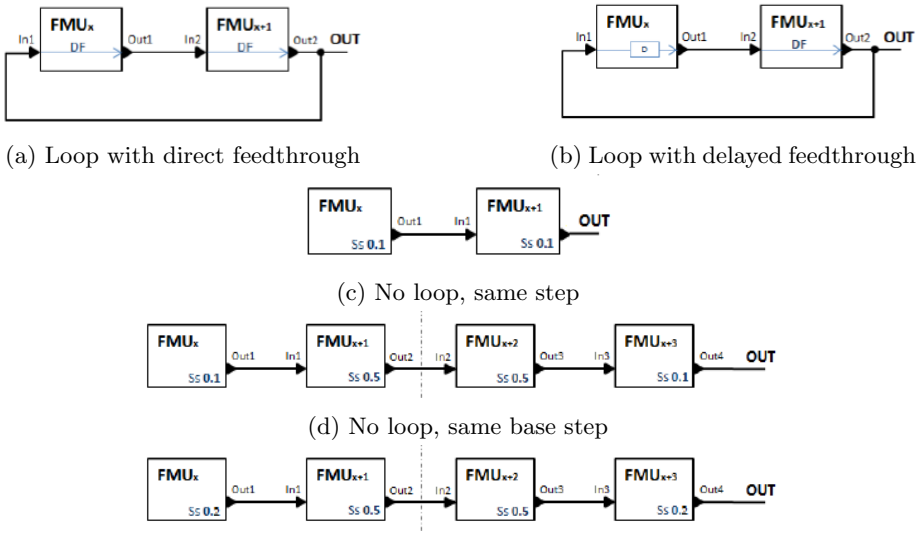
(Thule et al., 2018) mentions two main master algorithms, namely the Jacobi algorithm and the Gauss-Seidel algorithm. The Jacobi algorithm computes all the outputs from the subsystems first, then computes and sets the input. Then it simulates it to the next time point (*Figure 8a*). The Gauss-Seidel algorithm computes the input of the subsystems in the given order, then simulates the same subsystem to the next time point. It then takes that output and uses it to compute the input of the next subsystem in the order (*Figure 8b*).



(a) Jaccobi (b) Gauss-Seidel

Figure 8: Difference between Jaccobi and Gauss-Seidel algorithms (Thule et al., 2018)

(Van Acker et al., 2015) mentions that depending on the different co-simulation models, the master algorithm used is adapted. They identify five different scenarios, namely those containing direct feedthrough loops, those containing delayed feedthrough loops, single rate, same base step size, and different base step size (*Figure 9*). Feedthrough loop means that the data exchange starts and ends at the same FMU (Van Acker et al., 2015).



(a) Loop with direct feedthrough (b) Loop with delayed feedthrough

(c) No loop, same step

(d) No loop, same base step

(e) No loop, different base step

Figure 9: Five different scenario's of co-simulation models (Van Acker et al., 2015)

In the paper they assume that all FMUs support rollback (procedure to copy and later restore their complete state). This is used when there is a loop in the model. In case of same step size, the Gauss-Seidel algorithm is used.

For the same base step size, it depends if FMU_x step size is smaller or larger than the FMU_{x+1} step size. In case they are smaller, the exact multiple k is calculated. FMU_x is run k times before the FMU_{x+1} is run. The other way around, a linear interpolation is used to estimate the output of FMU_x.

The case of different base step size is similar to the same base step size. If FMU_x step size is smaller than FMU_{x+1}, then a step match, which is the execution of k-times FMU_x results in the correct output for FMU_{x+1} for a given simulation step. This might result in using extrapolation to find the output of FMU_x. The other way around, a linear interpolation is used.

2.4 Literature Study Conclusion

The conducted study on hybrid co-simulation, specifically using FMI, resulted in several papers using different approaches. Most of these approaches focuses on the communication between the heterogeneous models at the master algorithm level. The most popular approach for this seems to be FMI extension for time step.

There are two papers that stand out for us, namely the papers of (Sun et al., 2011) and (Camus et al., 2016). In both papers, there is an inhouse simulation tool, which is being expanded with third party models.

(Sun et al., 2011) proposes combining a specialized simulation tool with Modelica for co-simulation. They use Dynaplant which is a tool highly specialized for large fluid systems. The model library is quite restricted and the development of new Dynaplant model is time consuming. Saadida is their inhouse FMU simulation tool which contains solvers for integrating a given FMU. Dynaplant calls methods from this FMU simulator in order to manage data exchange and provide time step sizes for both simulation partners. This is an interesting approach, however, the simulation is done by a specialized simulation tool.

The paper of (Camus et al., 2016) also has an inhouse simulation platform, where they integrate FMU simulations in it. In the paper they use a wrapper for the FMU components within their system. This wrapper allows FMUs to easily interact with the other components within their system. This paper is related to the objective of this thesis. Namely, Safran wants to integrate different third party models within their platform and co-simulate them.

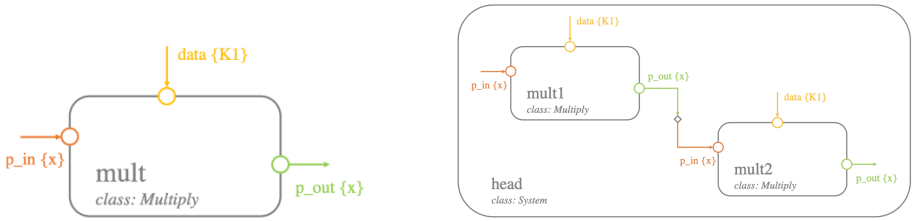
Thus, we have used the results of our literature study to select two papers that are most relevant to us. These papers are used as a reference for us when we are working on integrating third party models in Safran's platform.

3 Background Modeling and Simulation Platform

Before we can start to integrate third party models into Safran's simulation platform, we first need to understand how the simulation platform works. This section will explain how models are defined within the platform (*Section 3.1*), followed by how these models are currently simulated within the platform (*Section 3.2*). Lastly, we will explain the platform's shortcomings and how it can be improved (*Section 3.3*).

3.1 Models within the Modeling and Simulation Platform

The modeling and Simulation Platform is developed by Safran for the modeling of geometrical and functional shared models. Models that are developed within this platform are called systems. There are two types of systems, namely a simple system (*Figure 10a*), and complex systems (*Figure 10b*). Simple systems do not contain subsystems, while complex systems do.



(a) Simple System (b) Complex System

Figure 10: System in the Modeling and Simulation Platform

Systems have four types of variables, namely input, output, inwards, and outwards. The input is the input variables from external sources, and output is the output computed by the system. The inward is an input variable needed by the system to compute its output. The system in *Figure 10a* has an inward variable $data\{K1\}$, which is the number that the input of the system would be multiplied with. The outward is an intermediate output variable computed by the system that is used to compute the output. An example is as follows. Take the simple system of *Figure 10a* and imagine it would represent a multiplication in the form of $p\_in\{x\} * data\{K1\} = z$ and $z * data\{K1\} = p\_out\{x\}$. Then z would be an intermediate output. Thus z is the outward variable of this system.

Figure 11a shows an example of how a system class is declared in the platform. The inputs, outputs, inwards, and outwards of the system are defined in the *setup*. In the *compute*, the computation of the system is declared. *Figure 11b* gives an example of a system with subsystems. In this case, the subsystem and connections are declared in the *setup*. It also has no *compute* defined.

```python
class Pipe(System):
    """Poiseuille flow in a cylindrical pipe"""
    def setup(self):
        self.add_inward('D', 0.1, desc="Diameter")
        self.add_inward('L', 2.0, desc="Length")
        self.add_inward('mu', 1e-3, desc="Fluid dynamic viscosity")

        self.add_input(FloatPort, 'p1')
        self.add_input(FloatPort, 'p2')

        self.add_output(FloatPort, 'Q1')
        self.add_output(FloatPort, 'Q2')

        self.add_outward('k', desc='Pressure loss coefficient')

    def compute(self):
        """Computes the volumetric flowrate from the pressure drop"""
        self.k = np.pi * self.D**4 / (256 * self.mu * self.L)
        self.Q1.value = self.k * (self.p2.value - self.p1.value)
        self.Q2.value = -self.Q1.value
```

(a) Code snippet of a system *Pipe*

```python
class CoupledTanks(System):
    """System describing two tanks connected by a pipe (viscous limit)"""
    def setup(self, rho=1e3):
        self.add_child(Tank('tank1', rho=rho))
        self.add_child(Tank('tank2', rho=rho))
        self.add_child(Pipe('pipe'))

        self.connect(self.tank1.p_bottom, self.pipe.p1)
        self.connect(self.tank2.p_bottom, self.pipe.p2)
        self.connect(self.tank1.flowrate, self.pipe.Q1)
        self.connect(self.tank2.flowrate, self.pipe.Q2)
```

(b) Code snippet of a system with subsystems

Figure 11: Code snippets of system class

3.2 Simulation within Modeling and Simulation Platform

Within the platform there are systems with time dependent variables. These systems can be simulated using time drivers. Time drivers are timed simulations within the platform. These time drivers are used to perform simulations and co-simulations of the various systems defined in the platform. They are essentially the master algorithms of the simulation platform.

A simple system is simulated by returning the output of the system at each time step. This time step is defined by the user who created the system. A complex system requires the inputs and outputs of each subsystem within the system to be connected to each other first. Afterwards a co-simulation can be performed using the time drivers.

An example of such a simulation would be a system containing two tanks and a pipe. The two tanks contain a fluid and are connected together by a pipe. The idea is to simulate the equilibrium between the two tanks. *Figure 12* gives an overview of this system.

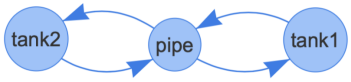


Figure 12: A system with the subsystems *tank*2, *pipe* and *tank*1

In this system, the outputs of *pipe* are connected to the inputs of *tank*1 and *tank*2, and the outputs of *tank*1 and *tank*2 are connected to the inputs of *pipe*. Using the time driver, a co-simulation can be performed on this system. The resulting graph can be seen in *Figure 13*. Each time step it can be seen that the height of the fluid in one tank is decreasing, while the height of the fluid in the other tank is increasing.

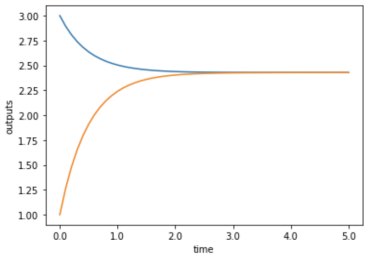


Figure 13: Result of the system with the subsystems *tank*2, *pipe* and *tank*1

3.3 Discussion of Modeling and Simulation Platform

In the previous sections we have given an overview of the Modeling and Simulation Platform developed by Safran. Users of this platform can develop models as systems within the platform. These systems can be simple or they can contain subsystems. They can also simulate the different systems they have developed within the platform.

The problem that Safran has is that the users of this platform can have a subsystem of the system developed in a third party platform. However, Safran's platform does not offer them a way to integrate this subsystem into their platform. Thus, they would want to allow the system to accept third party models.

4 FMU simulation in Python

In the previous section we have seen how the simulation platform developed by Safran works. Now we want to integrate third party models into this system. From the literature study we found that (Camus et al., 2016) had a similar problem as Safran. They integrate third party models into their system using the FMI standard. Safran's modeling and simulation platform's core is developed in Python. Thus, simulating FMUs in Python is essential.

There are several libraries in Python that allows the simulation of FMUs, including PyFMI (Modelon AB, 2018) and FMPy (Torsten Sommer, 2020). PyFMI uses a wrapper to access the functions of the FMUs. Safran's simulation platform will also require a wrapper to integrate FMUs within its platform. Thus having direct access to the FMU functions will ensure that the functions will not need to be encapsulated twice before being used in the platform. In the case of PyFMI, the new wrapper for Safran's platform would be dependent on changes made in the FMI standard and PyFMI wrapper. FMPy on the other hand lets you access the FMU functions directly. This means that the new wrapper for Safran's platform will only be dependent on changes made in the FMI standard. Thus, it was decided to use FMPy for this project.

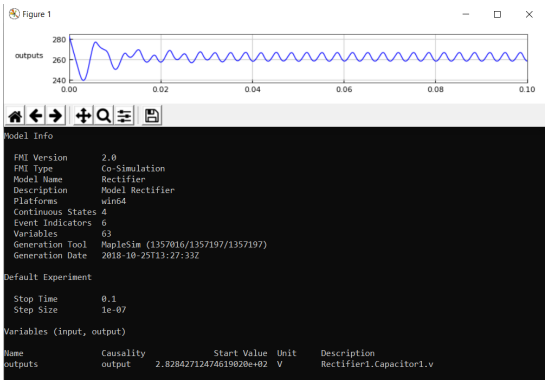
In the remainder of this chapter we will explain how we can simulate an FMU with FMPy and how we can extract information from an FMU using FMPy. We will run a co-simulation using FMPy and compare its performance using an existing co-simulation tool. Lastly, we will examine using different step sizes during a co-simulation of the same FMU.

4.1 Simulation with FMPy

To simulate FMUs in Python, FMPy was chosen. This is because FMPy lets you directly access the FMU code without going trough a wrapper. The simplest code to simulate an FMU is given in *Figure 14a*. From this code you get a simulation from the desired FMU. It can also give the model description which consists of the model type, stepsize, and inputs and outputs among other things (See *Figure 14b*).

```python
# Simple simulation of FMU with FMPy

from fmpy import *
from fmpy.util import plot_result

def sim_FMU(x):
    fmu = x
    dump(fmu) # show the FMU file
    result = simulate_fmu(fmu)
    plot_result(result)
```

(a) Code Snippet



(b) Resulting simulation of an FMU

Figure 14: An example of a simple simulation of FMU with FMPy.

However, the main idea is to connect multiple FMUs together and do a co-simulation (*Figure 15*). The code mentioned before will not be sufficient, and the examples given by FMPy are all simulations of one FMU.

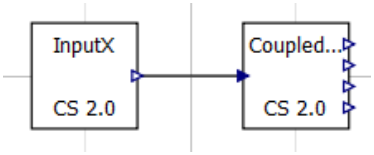


Figure 15: Example of a system with two FMUs to be Co-simulated

To simulate multiple FMUs, the input and output of the FMUs need to be extracted from the model description, as well as the step size of the FMUs. The code snippets in *Figure 16* shows the code we have developed to extract the aforementioned information from the model description using FMPy. The input and output are extracted directly from the model description while the step size is extracted from the default experiment in the model description.

```python
# read the model description
model_description = read_model_description(fmu_filename2)
model_description1 = read_model_description(fmu_filename1)

# collect the value references
vrs = {}
for variable in model_description.modelVariables:
    vrs[variable.name] = variable.valueReference

vrs1 = {}
for var in model_description1.modelVariables:
    vrs1[var.name] = var.valueReference

# get the value references for the variables we want to get/set
vr_inputs   = vrs['inputs']         # input CoupledClutches
vr_outputs = vrs['outputs[4]']      # output CoupledClutches
vr1_outputs = vrs1['y']             # output InputX
```

(a) Get inputs and outputs from the model description of FMUs

```python
ex1 = model_description1.defaultExperiment
exp1 = {}
exp1['stepSize'] = ex1.stepSize

start_time = 0.0
threshold = 300.5
stop_time = 1.5
step_size1 = exp1['stepSize']
```

(b) Get the step size of an FMU

Figure 16: Code Snippets with FMPy.

For the co-simulation of these FMUs, we had to use a while loop. While the stop time is not reached, the inputs and outputs are connected, both FMUs do a *doStep*, then the inputs and outputs value are extracted with the *get* function, and the time is advanced with the step size. A snippet of this code can be seen in *Figure 17*.

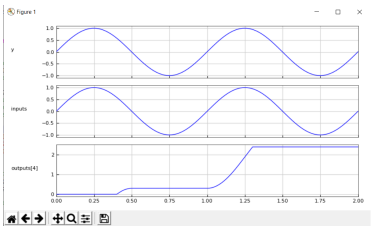
```python
fmu.instantiate()
fmu.setupExperiment(startTime=start_time)
fmu.enterInitializationMode()
fmu.exitInitializationMode()

fmu1.instantiate()
fmu1.setupExperiment(startTime=start_time)
fmu1.enterInitializationMode()
fmu1.exitInitializationMode()

time = start_time

rows = []  # list to record the results

# simulation loop
while time < stop_time:

    # set the input
    # must get the real value of the output
    fmu.setReal([vr_inputs], fmu1.getReal([vr1_outputs]))

    # perform one step
    fmu.doStep(currentCommunicationPoint=time, communicationStepSize=step_size)
    fmu1.doStep(currentCommunicationPoint=time, communicationStepSize=step_size)

    # get the values for 'inputs' and 'outputs'
    inputs, outputs = fmu.getReal([vr_inputs, vr_outputs])
    out1, = fmu1.getReal([vr1_outputs])

    # use the threshold to terminate the simulation
    if outputs > threshold:
        print("Threshold output4 reached at t = %g s" % time)
        break

    # append the results
    rows.append((time, out1, inputs, outputs))

    # advance the time
    time += step_size

fmu.terminate()
fmu.freeInstance()
```

Figure 17: Code snippet of simple co-simulation with fixed step size

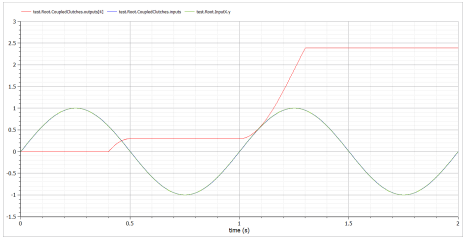
Figure 15 shows an example of a system containing two FMUs. The first FMU, $InputX$, is a simple FMU we have created using OpenModelica (OpenModelica, nd). Its output is a sinusoid. The second FMU, $CoupledClutches$, is an example FMU that is available from FMPy.

The idea is to compare the result of a co-simulation we generated using FMPy with the result of the same co-simulation using an existing co-simulation tool. For this reason we have opted to simulate the system described above using FMPy and OpenModelica. OpenModelica is an open-source Modelica-based modeling and simulation platform (OpenModelica, nd). One of the third parties' models that Safran wants to integrate within its platform is from Modelica. Since OpenModelica is a Modelica-based simulation platform, it is an appropriate tool to use.

The results of these two co-simulations were put next to each other and compared as can be seen in *Figure 18*. *Figure 18a* shows the output value of FMU $InputX$ as y, and the input and one of the output values of FMU $CoupledClutches$ as $inputs$ and $outputs[4]$ respectively. *Figure 18b* shows the output value of FMU $InputX$ as a green/blue line, and the input and one of the output value of FMU $CoupledClutches$ as a green/blue line and red line respectively. Here it can be seen that the output of $InputX$ is the same as the input of $CoupledClutches$ and is a sinusoide. In both co-simulations the output of $CoupledClutches$ is the same. In this way we validated that the generated co-simulation with FMPy gives the same result as when it would be generated with an existing tool.



(a) Plot Co-simulation in FMPy



(b) Plot co-simulation in OpenModelica

Figure 18: Plot of the co-simulation of FMUs $InputX$ and $CoupledClutches$

In conclusion, we were able to simulate an FMU in Python using FMPy. We were also able to co-simulate a system containing different FMUs within Python using FMPy. With this, we are one step closer to being able to co-simulate a system containing FMUs within Safran's platform.

4.2 Different step sizes

In the previous section, the simulation was done with a fixed step size. The question now arises what happens if two FMUs have different step sizes. *Figure 19* shows a system containing two FMUs. The first FMU, $Rectifier$, is an example FMU that is available from FMPy. *Figure 14b* shows the output of $Rectifier$. The second FMU, $CoupledClutches$ is the same FMU used in the system of *Figure 15*. $Rectifier$ has a step size of $1e^{-7}$, while $CoupledClutches$ has a step size of 0.01.

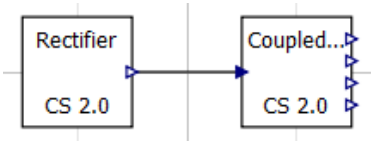
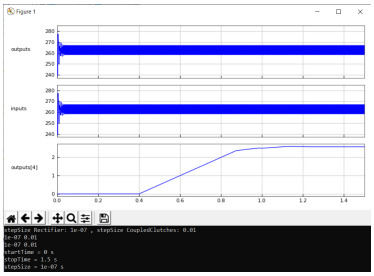
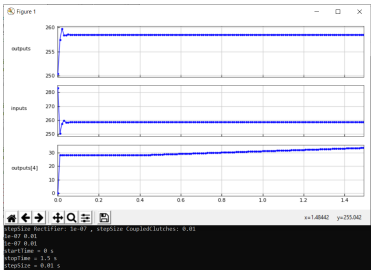


Figure 19: A System with two FMUs $Rectifier$ and $CoupledClutches$

We have co-simulated the system using FMPy twice. In the first co-simulation, we used the step size from $Rectifier$, namely step size $1e^{-7}$. In the second co-simulation, the step size from $CoupledClutches$ was chosen, namely 0.01. The results from these co-simulations can be seen in *Figure 20*.



(a) Plot co-simulation with step size $1e^{-7}$



(b) Plot co-simulation with step size 0.01

Figure 20: Difference between co-simulating FMUs with the smallest and largest step size

We know what the output from $Rectifier$ is supposed to be from *Figure 14b*. If we compare this with the output of $Rectifier$ in *Figure 20a*, it shows that the graph follows a similar damping sinusoide shape. However, if we compare it to the output of $Rectifier$ in *Figure 20b*, it becomes a constant instead of the damping sinusoid shape it originally had. Thus, from the results it can be seen that choosing the smallest step size between the two will give a more precise co-simulation.

However, choosing the smallest step size over the largest comes with its disadvantages. The algorithm has $O(n)$ complexity. Thus it takes much longer to compute the co-simulation using time step $1e^{-7}$ than it does using time step 0.01.

Another issue with using one over the other is the output of the second FMU *CoupledClutches*. The plots are giving different results (*Figure 20*). From our observations, we know that the output from $Rectifier$ in *Figure 20a* is more precise. Thus we can conclude that choosing the smallest step size between two FMUs is likely to be a better option with better results than choosing the larger step size. However, this does come with the price that it takes longer to compute. This becomes important later when we are looking at the Master Algorithm within Safran's platform.

5 Adjusting Modeling and Simulation Platform

So far, we have seen that with the help of the FMI standard, we are able to co-simulate different models as FMUs in Python using FMPy (*Section 4*). The objective of this project is to allow users of the modeling and simulation platform to complement their models (systems) with third party models, and support the co-simulation of various heterogeneous models. Thus, the next step is to integrate FMUs within the platform and to allow co-simulation with these FMUs. *Section 5.1* will explain how this can be done. Once this is possible, we can focus on the main question of this project; Can reusable composition operators between heterogeneous models be defined? And can we reify the Python based abstractions into Domain Specific Languages? These questions will be explored in *Sections 5.2* and *5.3*.

5.1 Integration FMU within Modeling Platform

The modeling and simulation platform has the same principle as FMUs, with the exception that FMUs are black boxes, while the systems in the platform are white boxes. As mentioned before, these models consists of four types of connections, namely input, output, inwards, and outwards (*Figure 10a*). The input and output corresponds to the inputs and outputs of FMUs.

The idea is to connect these systems together with external models, FMUs, and co-simulate them within the platform. *Figure 21* gives an example of a system consisting of three subsystems to be simulated. In this system, $fmu1$ is an FMU while *pipe* and $tank1$ are systems made within the platform.

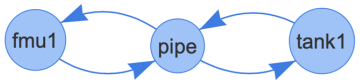


Figure 21: A system with the FMU $fmu1$, and systems *pipe* and $tank1$

To connect an FMU within the system, a wrapper for the FMU is needed. This will convert the inputs and outputs of the FMU into readable inputs and outputs of the platforms. The platform will consider each FMU as a system. Thus the purpose of this wrapper is to allow the FMU to conform to the format of a system in the platform. It will determine the input, output, inwards and outwards of the FMU. *Figure 22* gives a code snippet of the FMU wrapper we have developed.

```python
class FMUwrap(System):
    "FMU"
    def setup(self):
        self.add_inward('fmu_name', '/Users/janice/Documents/Python/CoupledClutc
        self.add_inward('threshold', 300.5, desc='threshold')
        self.add_inward('step_size', 1e-3, desc='stepSize')

        self.add_input(FloatPort, 'inputs_fmu')
        self.add_output(FloatPort, 'outputs_fmu')

        self.add_outward('fmu', desc='the fmu')
        self.add_outward('unzipdir', desc='extract the fmu')
        self.add_outward('fmu_time', desc='current time')
        self.add_outward('row', desc='row for the result')
        self.add_outward('vr_in', desc='input retrieved at each step')
        self.add_outward('vr_out', desc='output retrieved at each step')
        self.add_outward('model_description', desc='')
        self.add_outward('ex')
        self.add_outward('exp')
        self.add_outward('vrs')

        self.add_transient('x', der='vr_out')
        self.add_transient('y', der='vr_in')

        print('self.time {}'.format(self.time))
```

(a) Code snippet FMU wrapper

```python
def setup_run(self):
    self.model_description = read_model_description(self.fmu_name)
    self.ex = self.model_description.defaultExperiment

    self.exp = {}
    self.exp['tolerance'] = self.ex.tolerance
    self.exp['stepSize'] = self.ex.stepSize

    self.vrs = {}
    for var in self.model_description.modelVariables:
        self.vrs[var.name] = var.valueReference

    self.vr_in = self.vrs['inputs']
    self.vr_out = self.vrs['outputs[4]']

    self.unzipdir = extract(self.fmu_name)

    self.fmu = FMU2Slave(guid = self.model_description.guid,
                unzipDirectory = self.unzipdir,
                modelIdentifier = self.model_description.coSimulation.modelId
                instanceName = 'instance1')

    self.fmu.instantiate()
    self.fmu.setupExperiment(startTime = self.time)
    self.fmu.enterInitializationMode()
    self.fmu.exitInitializationMode()

    self.fmu_time = self.time
    print('setup: self.fmu_time {} self.time {}'.format(self.fmu_time, self.
```

(b) Code snippet FMU wrapper

```python
def compute(self):
    requested_time = self.time
    self.fmu.setReal([self.vr_in], [self.inputs_fmu.value])

    if requested_time > self.fmu_time:
        step = requested_time - self.fmu_time
        self.fmu.doStep(currentCommunicationPoint = self.fmu_time,
                        communicationStepSize = step)
        self.fmu_time += step

    self.x, self.y = self.fmu.getReal([self.vr_out, self.vr_in])
    self.outputs_fmu.value = self.x
    self.inputs_fmu.value = self.y


def clean_run(self):
    self.fmu.terminate()
    self.fmu.freeInstance()
    shutil.rmtree(self.unzipdir, ignore_errors = True)
```

(c) Code snippet FMU wrapper

Figure 22: Code snippets of FMU wrapper

As mentioned before, the FMU wrapper determines the inputs, outputs, inwards, and outwards of the FMU. In the code snippet it can be seen that this is done at the setup (*Figure 22a and 22b*). The input and output of the FMU wrapper are quite explicit. The input and output of the FMU are the input and output of the FMU wrapper. The inwards and outwards are less explicit. To determine the inwards, we look at the code for co-simulating FMUs from *Section 4.1*. The only input variable for this code is the FMU name. Thus this will be our inward variable.

As for the outward variables, we will look at all the helper variables we used in the code from *Section 4.1*. These consists of the variables for the model description, the inputs and outputs you retrieve at each step, and the step size among others. All of these will become the outward variables.

The FMU also needs to be initialized just like in the code from *Section 4.1*. Various variables also needs to be retrieved from the model description of the FMU. These are done during the setup_run.

During the compute, the wrapper connects the FMU input with the defined input of the wrapper (*Figure 22c*). This is done with the following code:

```
self.fmu.setReal([self.vr_in],[self.inputs_fmu.value])
```

The variable $self.vr\_in$ is the input from the FMU, and the variable $self.inputs\_fmu$ is the input from the FMU wrapper. It advances the time within the FMU for the given time step, using the *DoStep* function. This is the same code used for the simulation of FMU within Python using FMPy. It then sets the output of the FMU wrapper to the retrieved output of the FMU with the following code:

```
self.x = self.fmu.getReal([self.vr_out])
self.outputs_fmu.value = self.x
```

The variable $self.x$ is a transient, which allows the recording of a variable that is dependent on time. The variable $self.vr\_out$ is the output of the FMU, and the variable $self.outputs\_fmu$ is the output of the FMU wrapper.

Afterwards the FMU needs to be terminated (*Figure 22c*), just like in the code from *Section 4.1*. The wrapper does this in the clean_run with the *terminate*() function.

Now that we can use an FMU can be used within the platform with the help of the FMU wrapper, our next step is to connect it to the other systems within the platform. The code snippet in *Figure 23* shows that this can be done by adding an FMU wrapper as a subsystem to a system, and connect its input and output to the inputs and outputs of the other subsystems within this system. Then we can use the build in Master Algorithm, driver, to simulate the system. *Figure 24* shows the plot of the system after co-simulation.

```python
class CoupledTanks(System):
    """System describing two tanks connected by a pipe (viscous limit)"""
    def setup(self, rho=1e3):
        self.add_child(Tank('tank1', rho=rho))
        #self.add_child(Tank('tank2', rho=rho))
        self.add_child(FMUwrap('fmu1'))
        self.add_child(Pipe('pipe'))

        self.connect(self.tank1.p_bottom, self.pipe.p1)
        self.connect(self.fmu1.outputs_fmu, self.pipe.p2)
        self.connect(self.tank1.flowrate, self.pipe.Q1)
        self.connect(self.fmu1.inputs_fmu, self.pipe.Q2)
```

Figure 23: Code snippet of connecting FMU to other subsystems in a system

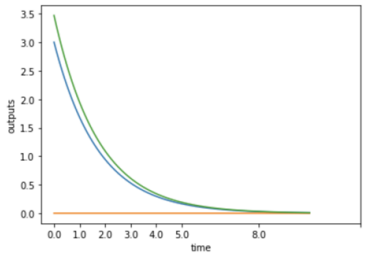


Figure 24: Plot of the system with the FMU $fmu1$, and models *pipe* and $tank1$

In conclusion, we were able to integrate an FMU within Safran's simulation platform. We achieved this by developing an FMU wrapper, which allows the FMU to conform to the format of the platform. With the help of this wrapper, we are also able to co-simulate a system within the platform containing FMUs. Thus we achieved the objective of allowing the modeling and simulation platform users to extend their various models (systems) with third-party models, and to support the co-simulation of the various heterogeneous models.

5.2 Generation of Python code using JSON

In the previous section we were able to allow the simulation of third-party models and the co-simulation of the various heterogeneous models. Now that we are able to do this, we can focus on the first research question;

1. Can reusable composition operators in between heterogeneous models be defined in a context of grey box co-simulation?

Each time a system is created, it consists of some basic operations. These operations include adding the subsystems to the system, add the connections between the subsystems, possibly define the execution order of the subsystems, and possibly define the initial values and constant values. *Figure 23* gives a snippet of this Python code. If we have the user defined information from the system as an input, it is possible to generate this piece of code. This input would contain the data structure of the system defined by the user. It does not contain the actual implementation of the system.

For the required input, we decided to use JSON. JSON is a text format that is completely language independent, easy for humans to read and write, and easy for machines to parse and generate (json.org, nd). The idea is that given a JSON schema, python code will be generated conforming to the systems in the modeling and simulation platform. The JSON schema is conforming to the schema standard of the platform, and is already specified. To get the Python code, a template for the system is used (*Figure 25*). This template will be used when generating the Python code.

```
import os
import json
from jinja2 import Template

# template for the system
SYSTEM_TMP = ('''"""System {{ name }}"""
```

Figure 25: Code Snippet of template for the system

A function *generatePythonFile* is declared that generates the Python file. It takes the information from the JSON file, and save it in a list of parameters called *param*. With this list we are able to make a distinction between subsystems, connections, inputs, and execution order. This list is then used to insert the correct information in the template for the system (*Figure 26*).

```python
# json file name, output folder, and output file name
fileName = 'system_config_pressureloss222.json'
output_folder = '/Users/janice/Documents/Python'
system_file = 'system_config_pressureloss222.py'

with open(fileName) as fileEx:
    json_data = json.load(fileEx)

# define generatePythonFile to generate the python file with the given json data
def generatePythonFile(j_data, out_folder, sys_file):
    sys = []

    for d in j_data:
        if '$schema' not in d:
            sys.append(d)

    # Get the class, inputs, connections, subsystems, and exec_order
    systeminfo = j_data[sys[0]]
```

Figure 26: Define *generatePythonFile*

Lastly, the template is written to a file (*Figure 27*). The file is saved at the given location and has the given name.

```python
# Write template to file
with open(os.path.join(out_folder, sys_file), 'w') as f:
    f.write(template.render(**params))
```

Figure 27: Write template to file

In the modeling and simulation platform it is possible that a system has a subsystem with its own subsystems (*Figure 28*). An example would be that if in the system in *Figure 21* with $fmu1$, *pipe*, and $tank1$, we replace $fmu1$ with the system $sys1$. This system would also consist of its own models that are connected to each other.

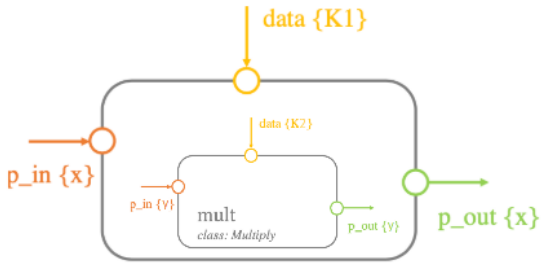


Figure 28: Recursively generate new template and file for each subsystem

For readability and practical reasons it is better to have all of these subsystems containing multiple subsystems of their own to be saved in separate files. This will also avoid one large file being updated when one of the subsystems evolve. The code will generated a new file for each subsystem containing multiple subsystems (*Figure 29*). When a system contains multiple subsystems, they are connected to each other. This connection is indicated through a connection in the JSON file. The function will recursively search for subsystems containing connections, and generate a new file for these subsystems.

```
# for all subsystems with connections, generate a new template and file
for s in systemsubsystems:
    if 'connections' in systemsubsystems[s]:
        s_data = { s : systemsubsystems[s]}
        o_folder = '/Users/janice/Documents/Python'
        s_file = system_file[:-3] + '_' + s + '.py'
        generatePythonFile(s_data, o_folder, s_file)
```

Figure 29: Recursively generate new template and file for each subsystem

The systems in the platform can also have an execution order. This order can be retrieved from the JSON file. However, it is not mandatory to specify an execution order, which means that it is not always present in the JSON file. In this case, the platform assumes the order the models are added into the system to be the execution order.

This way of computing the execution order can result in errors during co-simulation. An example would be a system like the one in *Figure 19*, where the execution order is given as $CoupledClutches$, $Rectifier$. $CoupledClutches$ needs the output of $Rectifier$ in order to compute its output, but it is being computed before $Rectifier$. Thus the system will not return the correct result. To avoid such situations, we have developed a code that computes the execution order using the topological order, specifically using Kahn's Algorithm (Kahn, 1962). If the system contains at least one cycle, then no topological order can be found. The cycles will be returned to the user. They can then decide to either remove the cycles from the system, or define an execution order themselves. *Figure 30* gives a snippet of the code we have developed for computing the execution order.

```python
# define topologicalOrder to calculate the execution order
def topologicalOrder(allModels, allConnections):
    # Empty list that will contain the sorted subsystems
    topological = []
    # Set of all subsystems with no incoming edge
    s = []
    incomingEdge = []
    for c in allConnections:
        incomingEdge.append(c[1])
    for m in allModels:
        if not any(m in e for e in incomingEdge):
            s.append(m)

    print(incomingEdge)

    # while s is not empty
    while len(s) > 0:
        # remove a subsystem from s and add to tail of topological
        n = s.pop(0)
        topological.append(n)
        # for each node c with an edge e from n to c
        for c in allConnections:
            # remove edge e from the graph
            print(c)
            print(incomingEdge)
            if n in c[0]:
                print('here')
                print(n)
                print(c)
                print()
                incomingEdge.remove(c[1])
                print(incomingEdge)
                # if m has no other incoming edge
                for m in allModels:
                    if m in c[1]:
                        if not any(m in e for e in incomingEdge):
                            # insert m into s
                            s.append(m)

    # If there are still incoming edges
    if len(incomingEdge) > 0:
        # graph has at least 1 cycle
        return None
    else:
        # return the topological order
        return topological
```

Figure 30: Execution order using topological order

Figure 31 Shows an example of a JSON file with its corresponding generated Python files. In the JSON file it can be seen that there is a system p that contains six subsystems, namely $p1$, $n1$, $p2$, $s1$, $p3$, and $p4$. Furthermore, it can be seen that subsystem $p2$ has a subsystems of its own, namely $p21$, $p22$, $s21$, $p23$, and $p24$. Subsystem $p22$ also has its own subsystems, namely $p221$ and $p222$. According to the described algorithm above, the file generation would produce three Python files, one for the system and one for each subsystem containing subsystems (p, $p2$, $p22$). *Figures 31c*, *31d*, and *31e* shows exactly this behaviour. Since the execution order is already present in the JSON file, it will not be computed during the file generation.

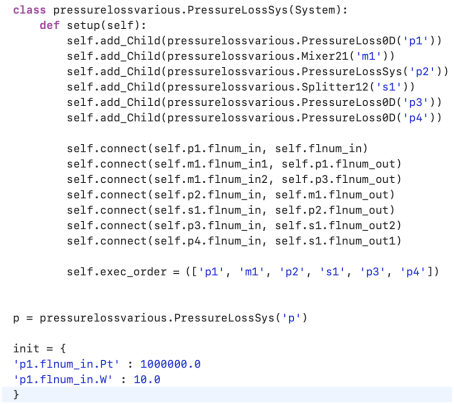
Thus, we were able to identify the reusable composition operators between the heterogeneous models. These include adding the subsystems, adding the connections between the subsystems, and the execution order of the subsystems. We were then able to use the data structure of systems defined by users as an input to generate the Python code containing these composition operators. We used JSON to represent the data structure of the systems.

```
{
  "$schema": "0-3-0/system.schema.json",
  "p": {
    "class": "pressurelossvarious.PressureLossSys",
    "connections": [
      ["p1.flnum_in", "flnum_in"],
      ["m1.flnum_in1", "p1.flnum_out"],
      ["m1.flnum_in2", "p3.flnum_out"],
      ["p2.flnum_in", "m1.flnum_out"],
      ["s1.flnum_in", "p2.flnum_out"],
      ["p3.flnum_in", "s1.flnum_out2"],
      ["p4.flnum_in", "s1.flnum_out1"]
    ],
    "exec_order": ["p1", "m1", "p2", "s1", "p3", "p4"],
    "subsystems": {
      "p1": {
        "class": "pressurelossvarious.PressureLoss0D",
        "inputs": {
          "flnum_in.Pt": 1000000.0,
          "flnum_in.W": 10.0
        }
      },
      "m1": {
        "class": "pressurelossvarious.Mixer21"
      },
      "p2": {
        "class": "pressurelossvarious.PressureLossSys",
        "connections": [
          ["p21.flnum_in", "flnum_in"],
          ["m21.flnum_in1", "p21.flnum_out"],
          ["m21.flnum_in2", "p23.flnum_out"],
          ["p22.flnum_in", "m21.flnum_out"],
          ["s21.flnum_in", "p22.flnum_out"],
          ["p23.flnum_in", "s21.flnum_out2"],
          ["p24.flnum_in", "s21.flnum_out1"],
          ["flnum_out", "p24.flnum_out"]
        ],
        "exec_order": ["p21", "m21", "p22", "s21", "p23", "p24"],
        "subsystems": {
          "p21": {
            "class": "pressurelossvarious.PressureLoss0D"
          },
```

(a) Code snippet of JSON file

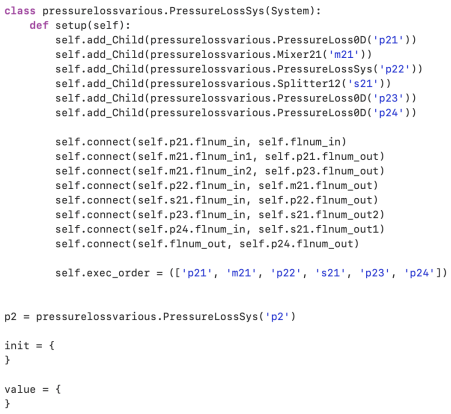
```
          "class": "pressurelossvarious.Mixer21"
        },
        "p22": {
          "class": "pressurelossvarious.PressureLossSys",
          "connections": [
            ["p221.flnum_in", "flnum_in"],
            ["p222.flnum_in", "p221.flnum_out"],
            ["flnum_out", "p222.flnum_out"]
          ],
          "exec_order": ["p221", "p222"],
          "subsystems": {
            "p221": {
              "class": "pressurelossvarious.PressureLoss0D"
            },
            "p222": {
              "class": "pressurelossvarious.PressureLoss0D"
            }
          }
        },
        "s21": {
          "class": "pressurelossvarious.Splitter12"
        },
        "p23": {
          "class": "pressurelossvarious.PressureLoss0D"
        },
        "p24": {
          "class": "pressurelossvarious.PressureLoss0D"
        }
      }
    },
    "s1": {
      "class": "pressurelossvarious.Splitter12"
    },
    "p3": {
      "class": "pressurelossvarious.PressureLoss0D"
    },
    "p4": {
      "class": "pressurelossvarious.PressureLoss0D"
    }
  }
}
```

(b) Code snippet of JSON file cont.



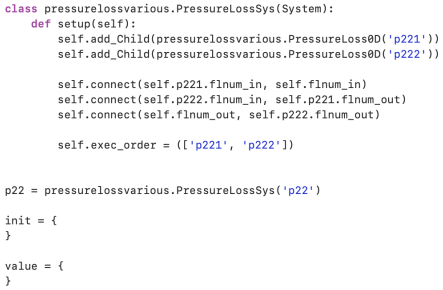
```
class pressurelossvarious.PressureLossSys(System):
    def setup(self):
        self.add_Child(pressurelossvarious.PressureLoss0D('p1'))
        self.add_Child(pressurelossvarious.Mixer21('m1'))
        self.add_Child(pressurelossvarious.PressureLossSys('p2'))
        self.add_Child(pressurelossvarious.Splitter12('s1'))
        self.add_Child(pressurelossvarious.PressureLoss0D('p3'))
        self.add_Child(pressurelossvarious.PressureLoss0D('p4'))

        self.connect(self.p1.flnum_in, self.flnum_in)
        self.connect(self.m1.flnum_in1, self.p1.flnum_out)
        self.connect(self.m1.flnum_in2, self.p3.flnum_out)
        self.connect(self.p2.flnum_in, self.m1.flnum_out)
        self.connect(self.s1.flnum_in, self.p2.flnum_out)
        self.connect(self.p3.flnum_in, self.s1.flnum_out2)
        self.connect(self.p4.flnum_in, self.s1.flnum_out1)

        self.exec_order = (['p1', 'm1', 'p2', 's1', 'p3', 'p4'])


p = pressurelossvarious.PressureLossSys('p')

init = {
'p1.flnum_in.Pt' : 1000000.0
'p1.flnum_in.W' : 10.0
}

value = {
}
```

(c) Code snippet of generated Python file



```
class pressurelossvarious.PressureLossSys(System):
    def setup(self):
        self.add_Child(pressurelossvarious.PressureLoss0D('p21'))
        self.add_Child(pressurelossvarious.Mixer21('m21'))
        self.add_Child(pressurelossvarious.PressureLossSys('p22'))
        self.add_Child(pressurelossvarious.Splitter12('s21'))
        self.add_Child(pressurelossvarious.PressureLoss0D('p23'))
        self.add_Child(pressurelossvarious.PressureLoss0D('p24'))

        self.connect(self.p21.flnum_in, self.flnum_in)
        self.connect(self.m21.flnum_in1, self.p21.flnum_out)
        self.connect(self.m21.flnum_in2, self.p23.flnum_out)
        self.connect(self.p22.flnum_in, self.m21.flnum_out)
        self.connect(self.s21.flnum_in, self.p22.flnum_out)
        self.connect(self.p23.flnum_in, self.s21.flnum_out2)
        self.connect(self.p24.flnum_in, self.s21.flnum_out1)
        self.connect(self.flnum_out, self.p24.flnum_out)

        self.exec_order = (['p21', 'm21', 'p22', 's21', 'p23', 'p24'])


p2 = pressurelossvarious.PressureLossSys('p2')

init = {
}

value = {
}
```

(d) Code snippet of generated Python file



```
class pressurelossvarious.PressureLossSys(System):
    def setup(self):
        self.add_Child(pressurelossvarious.PressureLoss0D('p221'))
        self.add_Child(pressurelossvarious.PressureLoss0D('p222'))

        self.connect(self.p221.flnum_in, self.flnum_in)
        self.connect(self.p222.flnum_in, self.p221.flnum_out)
        self.connect(self.flnum_out, self.p222.flnum_out)

        self.exec_order = (['p221', 'p222'])


p22 = pressurelossvarious.PressureLossSys('p22')

init = {
}

value = {
}
```

(e) Code snippet of generated Python file

Figure 31: Code snippet of JSON file and its generated Python files

5.3 DSL to generate JSON

In the previous section, we were able to answer our first research question;

1. Can reusable composition operators in between heterogeneous models be defined, in a contect of grey box co-simulations?

We are able to identify reusable composition operators between heterogeneous models. We use JSON to represent the data structure of the system to generate the Python code containing these composition operators. In this section we will focus on the second research question;

2. Can we reify the Python based abstractions into Domain Specific Languages to define such operations?

A JSON file was used to generate the desired Python code. However, as a user, it is not practical to write a JSON file each time you make a new system. Thus having a DSL to generate this JSON file is a better alternative. It is also possible to design a DSL which generates the Python code immediately. However, this intermediate step of generating a JSON file instead of the Python code immediately using the DSL ensures agility.

To define our DSL we will be using Xtext. Xtext is a framework for development of programming languages and Domain Specific Language (eclipse.org, ndb). It provides an infrastructure for you where you can write the syntax and semantics of your own language. It also has editing support for Eclipse and relies heavily on the Eclipe Modeling Framefork (EMF). This is a modeling framework and code generation facility for building tools and other applications based on a structured data model (eclipse.org, nda).

In this section we will first look at the syntax of our DSL (*Section 5.3.1*), followed the static semantics of our DSL (*Section 5.3.2*). Then we will give the implementation of our DSL (*Section 5.3.3*). Lastly, we will briefly give a discussion (*Section 5.3.4*).

5.3.1 Syntax

First we will look at what a system is, and which components it has. As was discussed before, we have two types of systems, namely a simple system and a complex system. Both systems have a name and a class. They can both have input ports and output ports. From the JSON file examples like the one in *Figure 31* , we can see that they can both have initial input values. Additionally, the complex systems can have subsystems, connections and an execution order.

Although there is a distinction between a simple system and a complex system, ultimately they are both systems. Furthermore, subsystems are also systems. Thus, we can say that a system has to have a name and a class, and can contain input and output ports, initial input values, systems, connections, and execution order. Thus, our language knows the following main concepts:

System The system which consists of a name, class, inputs, inports, outports, subsystems, connections, and execution order.

Input The input is either an initial input or a fixed input. This fixed input is used for the inwards of the system.

InitialInput These are the initial inputs of a system that are used for co-simulation.

FixedInput These are the inputs that correspond to the inwards of the system.

inPort These are the input ports of the system.

outPort These are the output ports of the system.

Subsystem A subsystem is a system.

Connection A connection is the connection between the subsystems of the system.

ExecOrder This is the execution order of the subsystems of a system.

Next to the main concepts of our language, we also have a few types that need to be addressed. These types are important for identifying the main concepts. What we mean is for example since a system has a name, we expect this to be of some type string. Our language knows the following types:

EString This type consists of all string variants. This means it is either a string or ID value.

ENumber This type consists of both integers and doubles.

EInt This type consists of all integers.

EDouble This type consists of all doubles.

Now that we have all the concepts and types defined, we will look at the syntax. *Figure 32* gives an overview of the syntax of our DSL. For readability we have opted to add keywords in front of a system name (*block*), and in front of a system class (:). Within the keywords { and } we can define the input and output ports, the initial input values, the subsystems, the connections, and the execution order. We have also opted to put keywords in front of input ports (*in*), output ports (*out*), and connections (*connection*). Initial input values are between brackets. All of these components are optional except for the name and the class. Furthermore, systems can contain zero or more inputs, subsystems, and connections.

We have defined input as either an initial input, or a fixed input. An initial input has the keyword '*initial*', followed by a name, a keyword '=' and an EString. Fixed input is the same as an initial input, however, it does not have the *initial* keyword. We define subsystems as a system. we define connection as two EString with a keyword '->' in between them. Lastly, we define an execution order as a keyword '*execution_order*' followed by a list of Estring.

```
grammar fr.inria.cosapp.sysGen.xtext.SysGen with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate sysGen "http://www.inria.fr/cosapp/sysGen/xtext/SysGen"

SystemProgram:
    systems=System;

System:
    'block' name=ID 'as' className=EString '('
    (inputs+=Input*)
    ')'
    ('{'
    ('In' (inPorts+=EString) (',' inPorts+=EString)*)?
    ('Out' (outPorts+=EString) (',' outPorts+=EString)*)?
    (subsystems+=Subsystem*)
    ('connections' '{' ((connections+=Connection) (',' connections+=Connection)*) '}')?
    (exec=ExecOrder)?
    '}' );

Input:
    InitialInput | FixedInput;

InitialInput:
    'initial' name=ID "=" ((intvalue= ENumber) | (value= EString))  ','?;

FixedInput:
    name=ID "=" ((intvalue= ENumber) | (value= EString))  ','?;

Subsystem:
    systems=System;

Connection:
    oSys=ID'.'output=EString '->' iSys=ID'.'input=EString;

ExecOrder:
    {ExecOrder}
    'execution_order' ((order+=EString) (',' order+=EString)*);


EString returns ecore::EString:
    STRING | ID;

ENumber:
    EInt | EDouble ;

EInt returns ecore::EInt:
    '-'? INT;

EDouble returns ecore::EDouble:
    '-'? INT'.'INT;
```

Figure 32: The syntax of our DSL in Xtext

Lastly, our language is not dependent on white spaces, line breaks, etc. Xtext has a default rule that allows white spaces, line breaks, etc. within the language. This rule is as follows:

```
terminal WS:
    (' ' | '\t' | '\r' | '\n')+;
```

You can disable this rule by parsing an empty set of terminals using *hidden*():

```
grammar org.eclipse.xtext.common.Terminals hidden()
```

You can alternatively pass an empty set of terminals using *hidden*() at the desired rules, for example:

```
ExecOrder hidden():
    {Execorder}
    'execution_order'
    ((order+=Estring)(',' order+=Estring)*);
```

However, we do not disable this rule in our language.

5.3.2 Static Semantic

In this section we will be looking at the static semantics of our language. We will first define the semantic rules of our language. Afterwards we will show how these can be implemented using Xtext.

The following rules can be defined for our language:

- The systems and subsystems should all have a unique name.
- The connection must be between subsystems.
- The execution order must contain only subsystems.

We will go through each of these rules and explain whether it is implemented in our language or not. In case we have not implemented a rule, we will explain how we can implement it.

The systems and subsystems should all have a unique name.
This rule has not been implemented in our language. In Xtext it is possible to have a unique name for all objects of a defined type. The defined type has to have a *name* variable that acts as the identifier of the object. Then we can uncomment the following line in the *.MWE2* file:

```
composedCheck = "org.eclipse.xtext.validation.
                    NamesAreUniqueValidator"
```

In our language, the subsystem has the type *System*. Thus by uncommenting this line, the names of the system and subsystems will be unique from each other.

The connection must be between subsystems.
We have not implemented this rule in our language. This can be done using cross-reference. Xtext allows you to specify the cross-references within the grammar. This can be done using the square brackets ([]), with the type or rule between the brackets. It is important to note that a terminal must be included if the rule is missing name ID. Otherwise it can be omitted. The following expression shows an example:

```
Connection:
    oSys=[Subsystem]'.'output=EString '->'
    iSys=[Subsystem]'.'input=EString;
```

Here both *oSys* and *iSys* are using the cross-reference to a *Subsystem*. Since *Subsystem* is of type *System* and has a *name*, it does not have to include a terminal.

The execution order must also contain the subsystems.
This rule has not been implemented in our language. It can also be implemented using the cross-reference. The following expression shows an example:

```
ExecOrder hidden():
    {Execorder}
    'execution_order'
    ((order+=[Subsystem])(',' order+=[Subsystem])*);
```

In this case, an object of *Subsystem* is added each time to the *order*.

In Xtext it is also possible to assign cardinality to expressions. There are four types of cardinalities, namely:

No operator means exactly one.

? This means zero or one.

***** This means zero or more.

+ This means one or more.

Furthermore, Xtext knows three different kinds of assignment operators, each with different semantics. These operators are as follows:

= This is used for features who only take one assignment.

+= This adds the value on the right hand side to the feature.

?= This expects a feature of type *EBoolean* and sets it to true if the right hand side was consumed.

We have implemented the cardinality and assignment operators within our language. In *Figure 32* we can see how we define *System*. The *name* is assigned an *ID* using the assignment operator =. This ensures that the name of the system has type ID. *className* is assigned with the operator = to *EString*. This ensures that the *className* has the type *EString*. We can also see the expression:

```
(inputs+=Input*)
```

This expression means that every time a value of type *Input* is added to *inputs*, and *inputs* consists of zero or more values.

On the next line of the syntax we see the expression:

```
('In' (inPorts+=Estring) (',' inPorts+=Estring)*)?
```

In this expression expresses that a value of type *EString* is added every time to *inports*, with a keyword of the character comma (,) between each value. Furthermore, the whole expression is either present or not, since it has a cardinality of zero or one (?). The remainder of the lines in the syntax for defining a System follow the same pattern. There is a list of values of some type that is continuously being added to the list, where the values may be separated by a keyword, and there is either zero or exactly one of this list.

We can also see in *Figure 32* how *Input* is defined. In this case we see the expression:

```
InitialInput | FixedInput;
```

This expression is telling us that an Input is either an *InitialInput* or (|) a *FixedInput*. The *InitialInput* is defined with the expression:

```
'initial' name=ID '=' ((intvalue=ENumber) |
(value=EString)) ','?
```

This expression starts with a keyword *initial*, followed by an assignment of *name* to a value of type *ID*. This ensures that *name* is of type *ID*. Then there is a choice between a *intvalue* being assigned a value of type *ENumber* or a *value* being assinged a value of *EString*. This ensures that an initial value is either of type *ENumber* or of type *EString*. It is then followed by an optional keyword of the comma character (,). The *FixedInput* is defined in a similar pattern as the *InitialInput*.

The subsystems is defined with the following expression:

```
system=System;
```

This expression is indicating that a value of type *System* is assigned to *system*. This essentially means that a subsystem is a system.

The execution order is defined with the following expression:

```
{ExecOrder}
'execution_order' ((order+=EString)
(',' order+=EString)*)
```

The {*ExecOrder*} in this expression ensures that an instance of *ExecOrder* is instantiated. Without this, the parser will be searching for an instance of *ExecOrder*, while instances of *ExecOrder* does not exist, and return an error. The second line of the expression is similar to other patterns we have covered before.

The *Connection*, *ENumber*, *EString*, *EInt*, and *EDouble* are also defined with their corresponding expressions. These expressions all follow similar patterns we have already covered before. Furthermore, Xtext automatically ensures that the language is not dependent on white spaces and new line break. This means that you can write the whole system on one line and it will see this as a valid system.

5.3.3 DSL

In the previous sections we have given the syntax and static semantics of our DSL. With this we can now declare a system within our DSL. *Figure 32* shows the syntax of our DSL written in Xtext. *Figure 33* shows an example of a system written in our DSL.

```
block system as CoupledTanks () {
    block tank1 as Tank (initial height = 3, area = 2) {
        In flowrate
        Out p_bottom
    }
    block tank2 as Tank (initial height = 0.8, area= 0.8) {
        In flowrate
        Out p_bottom
    }
    block pipe as Pipe (D = 0.07 , L = 2.5) {
        In p1, p2
        Out Q1, Q2
    }
    connections {
        tank1.p_bottom -> pipe.p1 ,
        tank2.p_bottom -> pipe.p2 ,
        tank1.flowrate -> pipe.Q1 ,
        tank2.flowrate -> pipe.Q2
    }
}
```

Figure 33: Example of a system written in the DSL

Now that we can define a system in our DSL, we can look at how we can generate the JSON file from this system. A feature in Xtext is the generator function. With this function, we can generate a file from our DSL. We use this to generate the desired JSON file. This file will have the same structure as the JSON files we have used before, for example in *Figure 31*. To do this, a template is defined where we will add the content of the JSON file. *Figure 34* shows the JSON file generated from the system written in *Figure 33*.

This generated JSON file can be used in the previously defined function *generatePythonFile* from *Section 5.2*. It is passed as our input to generate the desired Python files (*Figure 35*).

```
{
    "$schema": "0-3-0/system.schema.json",
        "system" : {
            "class" : "CoupledTanks",
            "connections" : [
                ["tank1.p_bottom","pipe.p1"],
            ["tank2.p_bottom","pipe.p2"],
            ["tank1.flowrate","pipe.Q1"],
            ["tank2.flowrate","pipe.Q2"]
            ],
            "subsystems" : {
                        "tank1" : {
                    "class" : "Tank",
                    "inputs" : {
                        "height" : 3,
                                    "inwards.area" : 2
                                    }
                },
                "tank2" : {
                    "class" : "Tank",
                    "inputs" : {
                        "height" : 0.8,
                                    "inwards.area" : 0.8
                                    }
                },
                "pipe" : {
                    "class" : "Pipe",
                    "inputs" : {
                        "inwards.D" : 0.07,
                                    "inwards.L" : 2.5
                                    }
                }
            }
                            }
}
```

Figure 34: Code snippet of generated JSON file from a system written in DSL

```
class CoupledTanks(System):
    def setup(self):
        self.add_child(Tank('tank1'))
        self.add_child(Tank('tank2'))
        self.add_child(Pipe('pipe'))

        self.connect(self.tank1.p_bottom, self.pipe.p1)
        self.connect(self.tank2.p_bottom, self.pipe.p2)
        self.connect(self.tank1.flowrate, self.pipe.Q1)
        self.connect(self.tank2.flowrate, self.pipe.Q2)

        self.exec_order = ['tank1', 'tank2', 'pipe']


system = CoupledTanks('system')

init = {
'tank1.height' : 3
'tank2.height' : 0.8
}

value = {
'tank1.inwards.area' : 2
'tank2.inwards.area' : 0.8
'pipe.inwards.D' : 0.07
'pipe.inwards.L' : 2.5
}
```

Figure 35: Code snippet of generated Python file from a system written in DSL

5.3.4 Discussion

We have identified the main concepts of our language with the help of what we consider a system. With the help of these concepts, we were able to develop the syntax for our language.

We have also given the static semantics of our language. Although the semantics are defined, not all of it is implemented in our DSL. We can still implement the unique names for the system and its subsystems. We can also implement that connections allow connections between subsystems, and execution order is only between the subsystems. Implementing these in the future will improve how our DSL works.

Our DSL lets us define a system. We can then generate the JSON file from it when saving a file containing a system. This JSON file is then used to generate the Python code containing the composition operators we have defines in *Section 5.2*. With this DSL we have answered our second research question.

5.4 Master Algorithm

So far we have only looked at integrating FMUs within the platform, and generating the Python code for a system using a DSL. As the literature study showed, the main focus of the papers is on the master algorithm that coordinates the co-simulation of the system itself. In the platform we call this master algorithm the drivers. From the literature study we can conclude that the most popular way to improve the master algorithm is by improving the step size of the system.

In this chapter we will discuss how the platform master algorithm works (*Section 5.4.1*), followed by how we can improve it (*Section 5.4.2*). Lastly, we will give a conclusion to our findings (*Section 5.4.3*).

5.4.1 Master Algorithm within Modeling and Simulation Platform

The platform initially used a very simple master algorithm. It was a simple while loop similar as the simple master algorithm in *Figure 17*. This means that it also used a fixed step size. However, in a later version of the platform, this code was updated by Safran.

The master algorithm still uses a while loop. However, it chooses the smallest step size between the different subsystems as the default step size. As we mentioned before in *Section 4.2*, using the smallest step size will give a more precise than using the larger step size. However, this does mean that the computation time is slower.

5.4.2 Future improvements Master Algorithm

The improvements done to the master algorithm by Safran already results in a better master algorithm. This is because it uses the smallest step size instead of a fixed step size. However, it can still be improved. As mentioned in *Section 1.1.3*, there are five different scenarios of a system, each with a different approach to solve them. *Section 1.1.3* also mentions the two main master algorithms, namely Jaccobi and Gauss-Seidel. Each of the different scenarios can use one of these master algorithms for improvements.

An example is implementing the Gauss-Seidel for systems with the same step size with no loops to use instead of the current while loop. Another example is implementing the Jaccobi algorithm for the scenarios that require rollback for interpolation or extrapolation.

Another point of improvement is to find a smarter way to calculate the step size instead of always using the smallest step size. One way is to use interpolation and extrapolation as described in *Section 1.1.3*.

5.4.3 Conclusion

The master algorithm of Safran's platform uses a simple while loop and the smallest step size to compute the co-simulation. With the help of existing papers and the literature study, we can safely say that Safran will be able to improve their co-simulation by improving their master algorithm with the two main master algorithms, namely the Gauss-Seidel and Jaccobi algorithms.

We have also seen that improving the step size of the master algorithm is also an important task for Safran. There are several papers that addresses this issue. They can use the five different scenario's as a baseline. Depending on which scenario the system is in, a different approach is carried out to calculate the step size.

6 Discussion

In this thesis we had the objective to allow third party models in the modeling and simulation Platform developed by Safran, and to co-simulate them. We have also asked two research questions, namely:

1. Can reusable composition operators in between heterogeneous models be defined, in a context of grey box co-simulation?
2. Can we reify the Python based abstractions into Domain Specific Languages to define such operators?

In this chapter we will discuss if we were able to achieve the objective (*Section 6.1*), and if we were able to answer research questions proposed (*Section 6.2*).

6.1 Objective

To allow third party models in the platform, we used the FMI standard. With the help of FMPy, we are able to access the different functions of the FMUs. A wrapper is then used to convert these functions into the format of the platform.

In this thesis we have taken a concrete example of an FMU and added a wrapper to it. When a different example is taken, most of the code would stay the same. The difference lies in the input and output of the FMUs. Setting up the correct inputs and outputs in the wrapper needs to be generated through reading the inputs and outputs from the FMU.

With the wrapper, we can now have various heterogeneous models in the platform. These can also be co-simulated using the time drivers. These time drivers are considered the master algorithms of the platform. As mentioned before, the master algorithm used in the platform uses a while loop for the co-simulation. A better alternative for this would be the Jaccobi or the Gauss-Seidel algorithm.

The master algorithm chooses the smallest step size, and uses it as the default step size. This might not always be desirable. A better way to do this is how (Bastian et al., 2011) approaches it. They simulate the FMU with a smaller step size the exact multiple times the larger step size, before simulating the FMU with the larger step size. Furthermore, we have conducted a literature study highlighting the increasing desire to improve the step size within co-simulations.

Our main objective was to allow third party models in the modeling and simulation platform developed by Safran, and to co-simulate them. As explained above, we were able to achieve this by using the FMU wrapper we have developed.

6.2 Research Questions

As mentioned before, we achieved the objective of allowing third party models in Safran's platform, and can co-simulate the various heterogeneous models. The connections between these heterogeneous models always consists of a *setup*. In this setup there is always a $self.add\_child$, a $self.connect$, and sometimes a $self.exec\_order$. These are the operators that we can reuse for the different systems. With the help of a JSON file as input, we are able to generate a Python file containing this code.

Thus, we were able to answer the first research question. As mentioned above, we were able to identify the different composition operators between the heterogeneous models. These composition operators are reused for different systems.

To get the JSON file needed for the input, we developed a simple DSL. Our DSL consists of *block* representing a system. Systems declared in this DSL will generate a JSON file, that is then used to generate the Python file.

We have opted to use the JSON file as an intermediate step due to the opportunity it presents when it comes to reusability and agility. Safran is looking into using a more graphic based approach to define systems. By having this intermediate step, it will be easier for them to integrate this visual DSL in the future with our existing code generator.

With this we have answered the second research question. We are able to reify the Python based abstractions into DSL to define the reusable composition operators. Our DSL takes the user input and is able to generate the Python file containing the code with the composition operators.

7 Conclusion and Future Work

In this thesis we looked at the modeling and and simulation platform developed by Safran. They wanted it to be able to allow third party models and simulate the various heterogeneous models within it. This was made possible using FMUs. A wrapper for the FMUs allowed them to correspond to the syntax of the platform. They can then be co-simuated using the time drivers (master algorithm) of the platform.

We have also looked at whether we can define reusable composition operators in between heterogeneous models, in the context of grey box co-simulation. It has been shown that the connection operators between the subsystems in a system in the platform can indeed be reused. With the help of a JSON file as input, we can generate the Python file containing these operators.

Lastly we have looked at whether we can reify the Python based abstractions into DSL to define these reusable composition operators. We defined a simple DSL to represent systems in the platform. The DSL then generate the JSON file. This file can then be used to generate the Python file.

In the future it is expected that Safran will want a visual way to declare systems. In this regard, the next step would be to design a DSL with these visual features. Another point is to add the logic and semantics into the json file. We would need to look more into the model to model transformations and model to text transformations, namely the logic of DSL to JSON transformation, and JSON to Python transformation. Safran also showed interest into adding time drivers into the JSON files. This is definitely a point of interest for the future.

The master algorithm (time drivers) currently being used by the platform can also be improved. The Gauss-Seidel and Jacobi algorithms can be implemented and compared to each other. An experiment with the different algorithms can be done to determine when it is better to use one or the other. Experimentation with getting better step sizes can also be a point of interest for the future.

References

Bastian, J., Clauß, C., Wolf, S., and Schneider, P. (2011). Master for co-simulation using fmi. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, number 63, pages 115–120. Linköping University Electronic Press.

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Elmqvist, H., Junghanns, A., Mauß, J., Monteiro, M., Neidhold, T., Neumerkel, D., et al. (2011). The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, number 063, pages 105–114. Linköping University Electronic Press.

Bogomolov, S., Greitschus, M., Jensen, P. G., Larsen, K. G., Mikučionis, M., Strump, T., and Tripakis, S. (2015). Co-simulation of hybrid systems with spaceex and uppaal. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 159–169. Linköping University Electronic Press.

Broman, D., Brooks, C., Greenberg, L., Lee, E. A., Masin, M., Tripakis, S., and Wetter, M. (2013). Determinate composition of fmus for co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 2. IEEE Press.

Camus, B., Galtier, V., and Caujolle, M. (2016). Hybrid co-simulation of fmus using dev&dess in mecsyco. In *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, pages 1–8. IEEE.

Cremona, F., Lohstroh, M., Broman, D., Di Natale, M., Lee, E. A., and Tripakis, S. (2016a). Step revision in hybrid co-simulation with fmi. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 173–183. IEEE.

Cremona, F., Lohstroh, M., Broman, D., Lee, E. A., Masin, M., and Tripakis, S. (2019). Hybrid co-simulation: it's about time. *Software & Systems Modeling*, 18(3):1655–1679.

Cremona, F., Lohstroh, M., Tripakis, S., Brooks, C., and Lee, E. A. (2016b). Fide: an fmi integrated development environment. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1759–1766. ACM.

Denil, J., Meyers, B., De Meulenaere, P., and Vangheluwe, H. (2015). Explicit semantic adaptation of hybrid formalisms for fmi co-simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 99–106. Society for Computer Simulation International.

eclipse.org (n.d.a). Eclipse Modeling Framework (EMF). Accessed 10 August 2020. From: `https://www.eclipse.org/modeling/emf/`.

eclipse.org (n.d.b). Language engineering for everyone! Accessed 10 August 2020. From: `https://www.eclipse.org/Xtext/`.

Feldman, Y. A., Greenberg, L., and Palachi, E. (2014). Simulating rhapsody sysml blocks in hybrid models with fmi. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 096, pages 43–52. Linköping University Electronic Press.

FMI (n.d.). Functional mock-up interface. Accessed 24 November 2019. From: `https://fmi-standard.org/`.

Galtier, V., Vialle, S., Dad, C., Tavella, J.-P., Lam-Yee-Mui, J.-P., and Plessis, G. (2015). Fmi-based distributed multi-simulation with daccosim. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 39–46. Society for Computer Simulation International.

Gomes, C., Karalis, P., Navarro-López, E. M., and Vangheluwe, H. (2017a). Approximated stability analysis of bi-modal hybrid co-simulation scenarios. In *International Conference on Software Engineering and Formal Methods*, pages 345–360. Springer.

Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. (2017b). Co-simulation: State of the art. *arXiv preprint arXiv:1702.00686*.

Jalali, S. and Wohlin, C. (2012). Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 29–38. IEEE.

json.org (n.d.). Introducing json. Accessed 3 July 2020. From: `https://www.json.org/json-en.html`.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

Lajolo, M., Lazarescu, M., and Sangiovanni-Vincentelli, A. (1999). A compilation-based software estimation scheme for hardware/software co-simulation. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)(IEEE Cat. No. 99TH8450)*, pages 85–89. IEEE.

Lee, E. A. (2008). Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE.

Modelica Association (n.d.). Modelica. Accessed 20 January 2020. From: `https://www.modelica.org/`.

Modelon AB (18 December 2018). Pyfmi. Accessed 17 February 2020. From: `https://pypi.org/project/PyFMI/`.

OpenModelica (n.d.). Introduction. Accessed 10 July 2020. From: `https://openmodelica.org/`.

Safran (n.d.). Safran at a glance. Accessed 7 January 2020. From: `https://www.safran-group.com/`.

Sun, Y., Vogel, S., and Steuer, H. (2011). Combining advantages of specialized simulation tools and modelica models using functional mock-up interface (fmi). In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, number 63, pages 491–494. Linköping University Electronic Press.

Tavella, J.-P., Caujolle, M., Tan, C., Plessis, G., Schumann, M., Vialle, S., Dad, C., Cuccuru, A., and Revol, S. (2016). Toward an hybrid co-simulation with the fmi-cs standard.

Thule, C., Gomes, C., Deantoni, J., Larsen, P. G., Brauer, J., and Vangheluwe, H. (2018). Towards the verification of hybrid co-simulation algorithms. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 5–20. Springer.

Torsten Sommer (23 May 2020). Fmpy. Accessed 17 February 2020. From: `https://pypi.org/project/FMPy/`.

Tripakis, S. (2015). Bridging the semantic gap between heterogeneous modeling formalisms and fmi. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 60–69. IEEE.

Van Acker, B., Denil, J., Vangheluwe, H., and De Meulenaere, P. (2015). Generation of an optimised master algorithm for fmi co-simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 205–212. Society for Computer Simulation International.

Wetter, M. (2011). Co-simulation of building energy and control systems with the building controls virtual test bed. *Journal of Building Performance Simulation*, 4(3):185–203.

Widl, E., Judex, F., Eder, K., and Palensky, P. (2015). Fmi-based co-simulation of hybrid closed-loop control system models. In *2015 International Conference on Complex Systems Engineering (ICCSE)*, pages 1–6. IEEE.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. Citeseer.

Zeigler, B. P., Muzy, A., and Kofman, E. (2018). *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press.