UMEÅ UNIVERSITY

FMIGo! A runtime environment for FMI based simulation.

Claude Lacoursière and Tomas Härdin

No 03/2018

Department of Computing Science

ISSN 0348-0542

Abstract

We present the software architecture of FMIGo!, a distributed, parallel runtime environment for executing Functional Mockup Interface (FMI) based simulations, describe how to use it, and how to extend its functionality. FMI is a standard used to define what Functional Mockup Units (FMU)s – or modules – can expose in terms of inputs and outputs, and the proper execution or call sequence one can perform on said FMUs. The FMI does not define the mathematics of modular time integration, a problem addressed by FMIGo! in an extensible way.

Also included are some theoretical and experimental aspects of different *stepping schemes*, or numerical methods to simulate coupled, modular systems. In particular, we present a *kinematic* solver which corresponds to Differential Algebraic Conditions between modules, as well as *wrapper* FMUs designed to augment the functionality of a given FMU with, for instance, input and output filters and directional derivatives, features rarely present in modules.

FMIGo! is open source under the MIT license and is meant to implement only the functionality required by steppers, as well as different stepping schemes.

Executive summary

What is FMI?

FMI is a standard which specifies an API, an XML Schema to describe models, and a state machine defining the behavior of Functional Mockup Units (FMU)s. The FMUs contain source code or executable binaries which implement a given model. FMUs are meant to be connected together to compose simulations out of blackboxes. A master stepper – not specified by the standard – can orchestrate the execution of a simulation by communicating data and commands to and from FMUs and advance simulation time.

Why should I be concerned with FMI?

Modeling and simulation of, say, robots and control algorithms is most often done using different software packages, or, at any given point, one might want to replace software models with hardware. The latter corresponds to Hardware in the Loop (HIL) and the former is Software In the Loop (SIL). As software tools are often incompatible, by design in many cases, the FMI can help if software tools can export FMUs which are then compatible.

For a controller algorithm created in Simulink and a robot model constructed with, say, Modelica, one can create FMUs and exchanging signals between the two modules is then simplified.

More briefly: FMI allows to couple simulations which were previously incompatible.

What is FMIGo! and why should I care?

FMIGo! is a run-time environment which can orchestrate FMI based simulations. It supports emerging standards for System Specification and Parameterization (SSP). It is distributed and parallel over both TCP/IP and the Message Passing Interface (MPI) so it can execute on clusters. It provides also a novel solver which can handle algebraic conditions between FMUs.

FMIGo! offers only core functionality and is free. This means that it provides a solid basis for user oriented tools. Because it is based only on protocols, it does not constrain the user to any data analysis program, authoring tool, etc.

Because development will continue in the academic context, the stepping algorithms are likely to evolve to include the latest developments in hybrid simulations, and be documented via scientific publications.

FMIGo! is released under the MIT license and is therefore provided AS IS, can be used for free in any context and for any purpose, even commercial. Solving initial algebraic loops requires linking with the GNU Scientific Library (GSL), in which case the resulting binary falls under the GNU General Public License (GPL). This feature is optional.

How do I test it?

First you must clone the `git` archive and follow the instructions for installation. There is no installer for Windows, though there are binary distributions. Compiling the code is not

completely fool-proof, especially under Windows. But this should be very easy on Linux and Mac.

There are very many examples provided which are used for testing. Creating examples can be difficult in one does not have SSP files. But the testing scripts should give a good idea of how to proceed.

Unless using the `pygo` interface, simulation data is available in `csv` format. There is also an interface to read data during the simulation. Examples are provided. The `pygo` module provides a simpler interface in `python` to specify small to moderately large simulations, and produces `HDF5` files, as well as scripts for data analysis and exploration.

What about the future?

At time of writing, the FMIGo! project is wrapping up and shutting down at Umeå university. However, Algoryx Simulation AB is looking into providing commercial support and committing changes back to main `git` archive. The first author will be involved in the mathematical development. Academics have expressed interest in implementing new simulation master algorithms.

In other words, one phase of this project draws to an end, but new developments are certain to come.

1 Introduction

There is great interest now to realize full-system simulations given the cost of building prototypes and the difficulty of tracing systemic issues in complex systems such as cars trucks and planes. Given that different subsystems are often modeled and using different software packages, there is a strong need for interfacing standards and numerical time integration methods.

The Functional Mockup Interface (FMI) is a standard which helps this at the first level. Several popular simulation software packages can export models as Functional Mockup Units (FMU)s which are zip files containing either binary, dynamically shared objects or source code implementing a given mathematical model, as well as an XML file describing the model, inputs, outputs, units and more. Two execution models are supported, namely, Model Exchange (ME), and Cosimulation (CS) as explained in Sec. 5. This makes it possible to easily share code or executable and connect different simulations which were never meant to be compatible. And that, in turn, opens the door for much broader functionality.

The standard includes the FMI API, the exact structure of FMU files, the XML schema, and "state machines" which describe the allowed execution sequences, meaning, what can happen at what stage in the simulation.

However, the standard specifies nothing with regards to the execution framework or the runtime infrastructure, or the numerical methods. It allows, in principle, the integration of continuous and discrete time simulations – hybrid systems – though common practice deviates from that as vendors offering *export* functionality often settle for the lowest common denominator as we discuss further in Sec. 2. There isn't a proper hybrid simulation stepper at this time so this is understandable behavior, yet frustrating from the perspective of numerical methods development since only the lowest common denominator is found in many FMUs.

FMIGo! addresses these problems by providing a foundation for FMI based simulations. And by "foundation", we mean a simple tool that performs a well defined task: read a configuration file, execute the simulation, and produce a data file which can then be imported for data analysis in various packages. We relied on standard libraries and protocols whenever possible, and widely available, portable libraries and languages otherwise. FMIGo! is not a commercial tool for integration, it is not a fully integrated piece of software. But it is designed to "play nice" with commercial products which handle fundamental aspects such as, say, live monitoring, data analysis, and the like.

FMIGo! is first an abstraction layer on the FMI API to allow for distribution over TCP/IP and MPI. Then comes the global stepper or "master". That part includes "signal routing" which reads from and writes to FMUs, and the implementation of time stepping algorithms, in particular, the one described in Sec. 14 which handles *kinematic* couplings, and is original. The global stepper can also resolve algebraic loops during initialization using Newton-Raphson iterations. A script is provided to parse System Specifications and Parameterization files (SSP)s – an emerging standard – and start the master.

The master module also provides a TCP/IP port to read the data in real-time during the simulation. An example for this is provided with the source code. It is of course also possible to include "monitor FMUs" which read data from other FMUs and plot this during execution.

Next is the *server* program which is responsible for loading *one* FMU, and as described in more details in Sec. 7. A simulation involves as many instance of the server program as there are FMUs.

Then we have adapters which can automatically augment modules with functionality needed for more sophisticated time stepping methods. This includes automated rollback for instance – when one needs to go back to a previous state as needed for error control – and directional derivatives which elicit the input-output relationships, or filters [3]. This is done with "wrapper FMUs".

Our motivation in writing this was to provide a solid runtime environment for the participants in the Virtual Truck And Bus (VTAB) project in cooperation with Scania, Algoryx, Volvo Cars, and Modelon. Our hope is that research groups will be able to use the software since there are few complete "import tools" in FMI, or those available may or may not do the job, or satisfy users' preference. A standalone open source project like this one can fill important gaps. Especially since it does not provide a full user environment but only specialized, though fundamentally important features directly related to the execution of a simulation. Our only effort towards a user interface is a set of cross-platform PYTHON scripts allowing the authoring of relatively simple simulations.

An important aspect of FMIGo! is that it does not provide a Graphical User Interface (GUI), or an authoring tool for editing simulations, and only supports conversion of output data to convenient formats, i.e., HDF5. It is intended to be a hub which connects well known tools via protocols, and allow seamless integration of well known tools, nominally incompatible. The concept follows the UNIX moto: "do one thing and do it well". This avoids vendor "lockin" and allows an ecosystem of loosely connected tools which exchange data via simple protocols leaving room for users to decide what they prefer.

In what follows we start by presenting the problem of coupling of discrete and continuous simulations in Sec. 2, follow up with a description of how systems are often modularized in Sec. 3 and explain why this can cause trouble in Sec. 4. This is followed by a brief description of FMI in Sec. 5, what it provides and what it doesn't provide, continue in Sec. 6 with the description of what a runtime or execution environment should provide, and how FMIGo! addresses this and provide some details of the architecture in Sec. 7. Then Sec. 10 describes one of the modules of FMIGo! which is essential to wrapper FMUs. The specifics of the cosimulation problem statement are covered in Sec. 11, and then Sec. 12 and Sec. 13 describe techniques for this. The kinematic stepper which offers an entirely different solution is covered in Sec. 14. Our `python` FMI support library, which serves as an interface to FMIGo! is presented briefly in Sec. 15. After which we conclude in Sec. 15.

2 Hybrid modeling and simulation

We first establish a few concepts of the theory of modeling and simulation before going to the specifics of the FMI and proceed to FMIGo! The focus here is on time domain simulations, and by this, we mean the numerical evaluation of the time evolution of dynamical systems which are approximations of given physical phenomena. These systems are assumed to be composed of *modules* and the goal is to combine these to produce reliable global simulations.

Dynamical systems can be represented by differential equations (DE)s in continuous time – a Differential Equation System Specification (DESS), or by Discrete Event Systems Specification (DEVS)s, or both [16]. An important subclass of DEVS are Discrete Time Systems Specification (DTSS). Modules can implement any of these paradigms internally. Hybrid sys-

tems are those which contain both discrete and continuous dynamical systems. The global stepper task is to implement a numerical method for the simulation, produce simulation data, and nothing else.

Since there are so many different types of DEs, we restrict the attention to explicit Ordinary Differential Equations (ODE)s with localized discontinuities – henceforth called "state events". This is the only type formally supported in FMI. This is written as

$$\dot{x} = f^{[i]}(x, u(t), t), \text{ subj. to } \quad z^{[i]}(x(t), u(t), t) \geq 0 \tag{1}$$

where x are the state variables, u are input variables, t is time, $f^{[i]}$ is the dynamical system's law which applies when the corresponding indicator function $z^{[i]}(x(t), u, t)$ is non-negative.

A state event is deemed to occur at $t_\star$ if $z^{[i]}(x(t_\star), u(t_\star), t_\star) = 0$. At which point, there can be discontinuous changes in the state variables $x(t_\star - \epsilon)$ and $x(t_\star + \epsilon)$ for arbitrarily small $\epsilon > 0$. To cross $t_\star$, the dynamical system must provide an impact law, and a transition from branch i to branch j in Eqn. (1).

As for DEVSS, they are black boxes with inputs u, state variables x, and outputs y, as well as one distinguished output variable, ϑ, which the closest future time at which an internal transition will occur if inputs are left unchanged.

At anytime other than $t = \vartheta$, changing the inputs on a DEVSS triggers an external transition which results in a new values of internal states x', ϑ', along with new outputs. Since $\vartheta' = \vartheta$ is a valid value, the inputs can immediately change the outputs without advancing time. This introduce superdense time $\mathcal{T}$ defined as the set of ordered pairs $\tau = (t, i), t \in \mathbb{R}$ and $i \in \mathbb{I}$, where t is the time and i labels *successive* events all at the same time t, such that $\tau \geq \tau'$ if $t \geq t'$ and $i \geq i'$. This introduces the global time index k meaning that if at step k we have superdense time $\tau_k = (t, l)$, then $\tau_{k+1} = (t, l + 1)$ if one of $\vartheta^{(j)}_{k+1} = t$, or we move to $\tau_{k+1} = (\min_j \vartheta^{(j)}_{k+1}, 0)$.

The representation of the update of a DEVS is then

$$(\vartheta_{k+1}, y_{k+1}) \leftarrow \Psi(\overset{k \leftarrow k+1}{\widetilde{x}}, \vartheta_k, u_k, \tau_k). \tag{2}$$

Given a collection of such modules $j = 1, 2, \ldots$, the simulation proceeds by sorting the different $\vartheta^{(j)}_k$, move to the smallest one, perform the internal transition, distribute inputs and outputs, trigger transitions to compute $\vartheta^{(j)}_{k+1}$, and start over..

There can be collisions if after an update to some time k we find that $\vartheta^{(i)}_k = \vartheta^{(j)}_k$ for any $i \neq j$. The resolution of this is dependent on execution order. This is beyond our scope at this time.

For DTSS, $\vartheta = \infty$ always. Outputs do not change in response to input changes until time is set to some value in the future, $t + H, H > 0$. The timestep (or communication step) H is chosen by either an error monitoring algorithm or by luck. Indeed, a special property of DTSS is that they might fail to reach $t + H$.

The most common representation of this is a specific discretization of a DESS which is then part of the black box. Mathematically, the time dynamics is represented as for the DEVSS except that we drop ϑ

$$y_{k+1} \leftarrow \Phi(\overset{k \leftarrow k+1}{\widetilde{x}}, u_k, t_k, H) \tag{3}$$

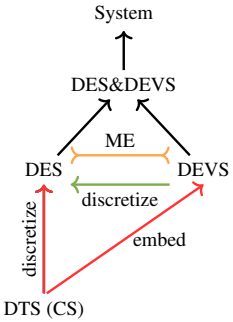

Figure 1: The hierarchy of system specification

where $H > 0$ is a time increment. The arrow over x represents that internal states are modified by applying an input u_k. After this point, $t = t_k + H$.

A hybrid simulation then includes all three kinds of system specification, which can be arranged in a hierarchy as shown in Fig. 2. Regrettably, FMI makes an artificial distinction in the API between DTSS and the combination of DESS& DEVSS which are considered as, respectively, CoSimulation (CS) and Model Exchange. This is not a problem for the stepping algorithm. However, vendors often support DTSS export only, even though their models would be better encapsulated as DEVS.

Algorithm 2.1 hybrid simulation

$k = 0, t = 0, \tau_0 = (0, 0)$, initialize
Find $\vartheta = \min_i \vartheta_k^{(i)}$
Integrate Eqn. (1) up to t (including event location)
Collect outputs y and update $u = l(y)$ for all modules
Set $\vartheta \leftarrow 0$
while $\vartheta = 0$ **do**
  for all j **do**

$$(\vartheta_{k+1}^{(j)}, y_{k+1}^{(j)}) \leftarrow \Psi^{(i)}(\overset{k \leftarrow k+1}{\widetilde{x}}, \vartheta_k^{(j)}, u_k^{(j)}, \tau_k)$$
$$u_k \leftarrow l(y_{k+1})$$

  end for
  $\tau_{k+1} \leftarrow \tau_k + (0, 1)$
  Find $\vartheta \leftarrow \min_i \vartheta_{k+1}^{(i)}$
  $k \leftarrow k + 1$
end while

When it comes to FMI, the ME API is intended to handle combinations of DESS and DEVSS in which the differential equations are expressed in a modular form, each module responsible to compute part of the derivatives of the state vector. The CS API in turn specializes to

coupled DTSS'. This arbitrary division is less than graceful and causes confusion, especially from vendors who feel compelled to choose one or the other paradigm and thus deprive the end users of needed features and limit the expressiveness of the simulations. But this is a different topic.

3 Modular modeling and simulation

Different software packages offer different functionality and coupling them is very useful. One might want to use package A to simulate the multibody dynamics of a vehicle, but software B to compute tire forces. Some formalism is required to abstract this process in order to construct a useful API.

We first focus on DESS to settle the notation and ignore state events for the moment. We consider a DAE of the form

$$\begin{aligned} \dot{x} &= f(x, u(t), t) \\ 0 &= y - g(x, u(t), t), \end{aligned} \tag{4}$$

where x are state variables, $u(t)$ are given inputs, and y are observable outputs. Event indicators are ignored for simplicity.

This is then partitioned this into subsystems $i = 1, 2, \ldots, n$ which have state variables $x^{(i)}$, inputs and output variables $u^{(i)}$ and $y^{(i)}$, respectively. Clearly, each of the input vectors $u^{(i)}$ contain elements of the original u vector, as well as elements of other state vectors $x^{(j)}$, $j \neq i$.

This leads to the representation

$$\begin{aligned} \dot{x}^{(i)} &= f^{(i)}(x^{(i)}, u^{(i)}, t), \quad i = 1, 2, \ldots, n \\ y^{(i)} &= g^{(i)}(x^{(i)}, u^{(i)}, t), \quad i = 1, 2, \ldots, n \\ u &= l(y), \end{aligned} \tag{5}$$

where u and y are the concatenation $u^{(i)}$ and $y^{(i)}$, respectively. Ignoring the third equation $u = l(y)$ which is to be implemented by a global stepper of sort, we obtain causal "black boxes"

$$\begin{aligned} \dot{x}^{(i)} &= f^{(i)}(x^{(i)}, u^{(i)}) \\ y^{(i)} &= g^{(i)}(x^{(i)}, u^{(i)}). \end{aligned} \tag{6}$$

Viewed like this, each module appears to have given inputs $u^{(i)}$ and return outputs $y^{(i)}$. The global condition $u = l(y)$ can be loop free if $\partial y^{(i)}/\partial u^{(i)} = 0$, otherwise, it is a nonlinear equation in u which must be resolved iteratively. In any case, enforcing the condition $u = l(y)$ is not part of any individual module. Note also that, despite the fact that the DAE in Eqn. (4) is of index 0, the modular representation allows the representation of DAEs of higher order when there are algebraic loops, i.e., when $\partial y/\partial u \neq 0$.

Example 3.1 (Simplest case)**.** *Consider the second order autonomous system defined by*

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = f(x) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_2 \\ f_2(x_1, x_2) \end{bmatrix} \tag{7}$$

where the subscripts $(\cdot)_i$ denote vector components (this will denote discrete time elsewhere). To frame this in the notation of Eqn. (5) *we need*

$$\begin{aligned} x^{(1)} &= x_1, \quad u^{(1)} = x_2, \quad f^{(1)}(x^{(1)}, u^{(1)}) = u^{(1)} \\ x^{(2)} &= x_2, \quad u^{(2)} = x_1, \quad f^{(2)}(x^{(2)}, u^{(2)}) = f_2(u^{(2)}, x^{(2)}) \end{aligned} \tag{8}$$

Which leads to

$$\begin{aligned} \dot{x}^{(1)} &= u^{(1)} \\ \dot{x}^{(2)} &= f^{(2)}(x^{(2)}, u^{(2)}) \\ y^{(1)} &= g^{(1)}(x^{(1)}, u^{(1)}) = x^{(1)} \\ y^{(2)} &= g^{(2)}(x^{(2)}, u^{(2)}) = x^{(2)} \textit{ and} \\ l(y) &= \begin{bmatrix} u^{(1)} \\ u^{(2)} \end{bmatrix} = \begin{bmatrix} y^{(2)} \\ y^{(1)} \end{bmatrix} = \begin{bmatrix} x^{(2)} \\ x^{(1)} \end{bmatrix} \end{aligned} \tag{9}$$

Clearly, we transformed an ODE into a DAE. Note however that $\partial y/\partial u = 0$ so we do not have algebraic loops.

When state events are added, we need to assume first that no two transitions are coïncidental or that they can be resolved safely in sequence. Writing $f^{(j,l_j)}$ and $z^{(j,l_j)}$ for the current branch of module j, a numerical time integration method then proceeds with $\dot{x}^{(j)} = f^{(j,l_j)}$ as long as $z^{(j,l_j)}(t) > 0 \forall j$, but when detecting that $z^{(j,l_j)}(t) < 0$ for at least one j, binary search is applied in time until sufficiently close to $z^{(j)}(t_\star) \approx 0$, at which point we resolve the event and transit to $f^{(j,l'_j)}$, keeping all other $f^{(i,l_i)}, i \neq j$ fixed. Here lies the problem though, that two nearby transitions might be resolved out of order depending on machine precision and conditioning. Superdense time is not considered in this formalism which is why the FMI standard specifies two modes of execution in ME, namely, continuous mode with continuous time, and event mode for superdense time.

4 System splitting

The previous section introduced the notation and the basic notion of modularity. But that leaves the question of how to modularize of a dynamical system which does not consist of independent components, i.e., is defined from Eqn. (4). Starting from an autonomous system $\dot{x} = f(x)$ for simplicity, we might choose to group the variables into $x^{(1)}$ and $x^{(2)}$. The equations of motion then read

$$\begin{aligned} \dot{x}^{(1)} &= f^{(1)}(x^{(1)}, x^{(2)}) \\ \dot{x}^{(2)} &= f^{(2)}(x^{(2)}, x^{(1)}) \end{aligned} \tag{10}$$

which we can now write *formally* as

$$\begin{aligned} \dot{x}^{(1)} &= f^{(1)}(x^{(1)}, u^{(1)}) \\ \dot{x}^{(2)} &= f^{(2)}(x^{(2)}, u^{(2)}) \\ y^{(1)} &= x^{(1)}, \quad y^{(2)} = x^{(2)} \\ u^{(1)} &= y^{(2)}, \quad u^{(2)} = y^{(1)}. \end{aligned} \tag{11}$$

However, this does not mean that we can integrate the two in parallel: system (1) needs the values of system (2) at all stages of a numerical time integration method. If $u^{(i)}(t)$ is constant, we will make serious numerical errors and essentially, be restricted to first order only. Though there are specialized partitioning methods where different integrators can be used for each of the subsystems [6], which splits the computations and allows some parallelism, this is very restricted in scope and not applicable to the general case.

Finding ways to simulate the two systems in parallel with signal exchanges at communication intervals H, larger than the *natural* time step used for the subsystems is the central problem of cosimulation.

The temptation to rewrite Eqn. (4) as Eqn. (11) arises whenever $\partial f^{(i)}/\partial x^{(j)}$ has very few non-zero entries so that in fact, we should write

$$\begin{aligned} \dot{x}^{(1)} &= \tilde{f}^{(1)}(x^{(1)}, u^{(1)}) \\ \dot{x}^{(2)} &= \tilde{f}^{(2)}(x^{(2)}, u^{(2)}) \\ y^{(1)} &= P^{(1)}x^{(1)}, \quad y^{(2)} = P^{(2)}x^{(2)} \\ u^{(1)} &= y^{(2)}, \quad u^{(2)} = y^{(1)}. \end{aligned} \tag{12}$$

where $P^{(i)}$ are projection operators, and

$$f^{(i)}(x^{(i)}, x^{(j)}) = \tilde{f}^{(i)}(x^{(i)}, u^{(j)}). \tag{13}$$

We therefore need some kind of extrapolation for $u^{(i)}$, or a modification of $\tilde{f}^{(i)}$ so that

$$\begin{bmatrix} \dot{x}^{(i)} \\ \dot{u}^{(i)} \end{bmatrix} = \begin{bmatrix} \tilde{f}^{(i)}(x^{(i)}, u^{(i)}) \\ \hat{f}^{(i)}(x^{(i)}, u^{(i)}), \end{bmatrix} \tag{14}$$

and we assume now that $\partial \hat{f}^{(i)}/\partial x^{(i)}$ has very few non-zero entries, meaning that the new system is still easy to integrate.

But some cases are much more difficult in part because they aren't defined by ODEs but instead, have algebraic conditions as would be the case in a mechanical system

$$\begin{aligned} \ddot{x} &= f(x, \dot{x}, u) + C^T\lambda \\ c(x) &= 0, \text{ where } C = \frac{\partial c}{\partial x}, \end{aligned} \tag{15}$$

and here, the temptation would arise if C had very few non-zero columns, and just a few rows.

For such cases, the modularization should also include a global algebraic condition

$$c(x^{(1)}, x^{(2)}) = 0, \tag{16}$$

to be enforced at each communication step.

Therefore, the goal of cosimulation is to either find a way to enforce Eqn. (16) at a global level, or find functions $\bar{f}^{(i)}$ which work sufficiently well. The latter case will be used in Sec. 12 and Sec. 13, and the former in Sec. 14.

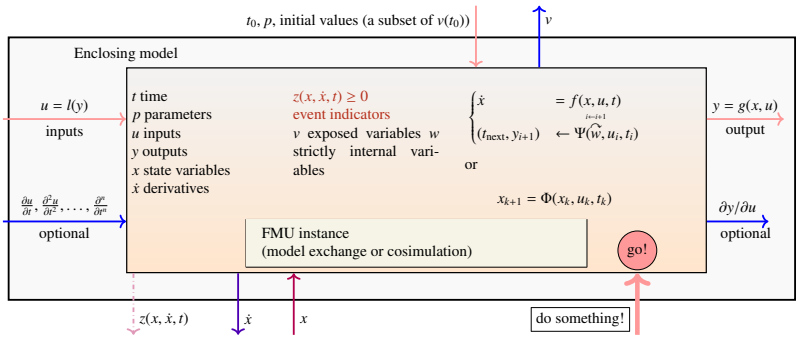

Figure 2: The black box model

5 Summary of FMI

The Functional Mockup Interface (FMI) is a collection of an API representing a minimal – though increasing – set of operations one should expect or would want from individual modules, an XML Schema to describe the capability and variables exposed by a module, a format specification for how to package a module, and state machines describing which operations are allowed in what state or at which stage of the simulation.

The fundamental concept is described in Fig. 2, a simple black box. We highlighted the z variables as they do not appear in the original FMI documentation but are implied. The FMI distinguishes between three types of black and gray boxes, namely, those which have continuous states and possibly event indicators, DESS, and those which do not have continuous states are split between DEVS which have a *time to next event* functionality, and DTSS which are just discrete time systems.

Vendors or software developers are expected to *export* their code in source or binary form – as shared objects – in a zip file, which is the concrete realization of a Functional Mockup Unit (FMU), which we often refer to as *module* when the details of the standard aren't relevant to the discussion. Runtime environment are expected to *import* FMUs with dynamic loading, and perform numerical time integration on said FMUs.

Requirements on the runtime environment and time stepper are scant except that the state machine specifications are expected to be respected – which they aren't in practice however. There are no compliance tests defined for this. This is both liberating and constraining since vendors may or may not realize the usefulness of exporting FMUs in a particular way, since they cannot know what to expect from the runtime environment. The smallest viable functionality is often the choice, and time stepping schemes take considerable liberty.

The FMI is very nearly the lowest possible denominator when it comes to interfacing to existing pieces of software which implement DEVSS or DESS or DTSS, and execute said in a hybrid simulation. A regrettable distinction was made however so that exported FMUs are categorized as Model Exchange (ME) if they are DESS&DEVSS combinations, or CoSimulation (CS) for DTSS, at least in the mind of those exporting FMUs. As discussed in Sec. 2, DTSS are subsets of DEVSS and should not really have a separate API. Instead of having to

decide between ME DEVSS or and CS DTSS, one should simply decide what to implement in ME DEVSS. But this is the subject of a deeper analysis.

In one sense, the FMI specification is an upgrade on Simulink which has been de-facto standard for tool integration for some time already. This is because the FMI clearly specifies requirements on modules, unlike the case of S-functions in Simulink.

A glaring issue is the lack of clear specification for Differential Algebraic Equations (DAE)s which is truly disappointing. This would have provided a true upgrade to Simulink. However, given the nature of the black box specification, it is possible to define DAEs as modules which always reports zero time derivatives, but then, one needs a specialized stepper.

The next section describes the obvious – though not standardized – requirements on an import tools such as FMIGo! and the solutions we developed.

6 Runtime environment requirement

Given the lack of specification for import tools such as FMIGo!, there is both great latitude for implementing functionality, yet technical difficulty in the realization of a full featured runtime environment. There are multitudes of global stepper implementations, but they implement only minimal functionality. Yet something solid is needed in order to test numerical methods.

The first issue is that there aren't good guidelines for the implementation of the FMI API. However, Qtronix provides an open source library with many required features including logging, error checking and so on and so forth. But more importantly, there is nothing at all for supporting parallel implementation over TCP/IP or MPI or other communication protocols. This despite the fact that such runtime environments are specifically mentioned and suggested in the standard's documentation. We had to resort to our own `protobuf` implementation for the messaging format, and our own ZeroMQ implementation for messaging over TCP/IP, likewise, our own design of the MPI implementation. The fact that this prevents interoperability of different implementations of clients and server is mild, though this leads to duplication of tedious, time consuming work. The FMI committee is in fact looking at standardizing this protocol. This said, we have found that `protobuf` incurs a performance hit because of excessive number of operations related to copying, packing, and unpacking messages. This is being revised at time of writing.

In addition FMI has no provision for the practical representation of a full simulation, i.e., the list of connections between the modules, or how and where to find specific FMUs. A simple URI specification would have been good. Likewise, tools are needed to author these simulations. Fortunately there is a standard emerging for defining a full simulation, namely, the System Specification and Parameterization (SSP). This includes both an XML schema for the list of connections, and a file format to package all the FMUs needed for a given configuration. We settled for that standard that will make FMIGo! be much more useful and available, especially given the emergence of SSP editors. This allowed us to completely ignore the authoring components, beyond simple Python modules for simple simulations which can be specified programmatically.

Sorely missing from the FMI are vectors and matrices specification. The onus then relies on the runtime environment to provide efficient ways to communicate such data. Yet we have not spent effort on this yet as the FMI committee has promised such features in the future.

Also missing is the specification of the output data format from simulation. Given that one of the implied functionality of FMI is to be able to implement a module with different FMUs, this poses a challenge in data analysis as one needs alias definitions which map equivalent variables names across modules. SSP answers some of these issues, but not at the data analysis level. We've addressed this problem in our PYTHON modules, but not at the SSP level yet.

Different import tools of course implement different time integration methods and we put considerable effort in developing good techniques for this. In particular, we implemented a stable kinematic stepper for CS FMUs which then allows for algebraic conditions in Sec. 14.

Since there are times when derivatives are often needed for robust algorithms and yet not often implemented by vendors, we also set to augment FMUs with such functionality as numerical derivatives, and likewise, to augment ME FMUs so they support CS features which are straightforward to implement automatically. This was done with *wrapper FMUs* which are automatically generated.

Likewise, special techniques such as filtering [3] can stabilize DTSS simulations though generally, these require a DESS to start with so the filter can be applied during the time integration of the differential equations. Since there are many cases where DESS export is feasible yet the global simulation is composed of DTSS, we automated such features wrapper FMUs as well. Of course, using this requires coercing modelers in exporting ME FMUs instead of default CS.

We have even considered wrapper FMUs to implement brute force methods to provide rollback and sync capability. The former is needed for iterative methods such as our kinematic stepper, the latter is needed to do local error analysis where one performs two simulation simultaneously as in the case of Richardson extrapolation, which requires sync and reset [15]. This has not succeeded yet.

To summarize, FMIGo! implements

- protocols to encapsulate the FMI API
- implementation of distributed execution over TCP/IP and MPI
- realtime data piping between the simulation master and "monitors"
- parsing of SSP files
- wrapper FMUs to augment the feature set of ME FMUs.
- scripting tools for authoring and analyzing simple simulations.

7 Software architecture

FMIGo! is a distributed client-server architecture. The client here is the global stepper – since it consumes data – and the servers handle the FMUs. Each server is a separate process which prevents symbol clashes between shared library. The servers can live on different computers and users are expected to use this to protect Intellectual Property (IP).

Servers can also perform numerical differentiation when needed, for FMUs that do not provide such functionality. Local numerical integration however can only be done by wrapping ME FMUs using the wrapper scripts mentioned in Sec. 6.

The wrapper system itself is based around a set of `python` scripts and C code making use of the Qtronic FMI library [14] and GNU GSL. These scripts and C code are automatically invoked/built as needed via some `CMake` plumbing.

Servers unzip FMUs, parse the model description files, dynamically load the executable, wait on the client to start the simulations, and convert messages from the FMI API to the communication layer.

The client performs the time stepping and communicates with the servers to read and write data to respective FMUs. This was implemented in `C++` and the class hierarchy are designed so that it is relatively easy to introduce new steppers. The master stepper can also resolve algebraic loops during initialization using GNU Scientific Library's *hybrids* multiroot solver.

The global concept is demonstrated in Fig. 3. Of particular notice is that we view FMIGo! as a minimal component on which commercial or academic solutions should be built. This decisions was motivated primarily by the UNIX philosophy: do one thing and do it well. Also with the firm belief that some code such as UIs should not be developed at the university. That's what software companies are for.

The server construction is illustrated in Fig. 4. As said, these are meant to handle both ME and CS FMUs, and in some cases, convert ME to CS, providing additional features.

As for the main stepper shown in Fig. 5, the diagram illustrates a modular construction which permits several numerical methods to be added, but also mixed and matched. We do not have full hybrid systems capability at this time, however, but only our kinematic stepper as well as common schemes which only pass signals between FMUs with user specified sequential or parallel order, or a mix of both.

This design is meant to be modular and separate the computations from the main application. Given the licensing model, it is possible to hide FMUs behind firewall with just a few ports open allowing instantiation and communication with the hidden FMU, thus providing IP protection. And because of the license, it should be possible for academics to try new stepping methods and benefit from the rest of FMIGo!

Also, all visualization components are left open. An application designer is expected to use a communication port from the master to extract data which can be plotted during execution, or control of the application e.g., start, pause, stop. As shown in Fig. 3, everything to do with GRID management, model management and configuration authoring is left to other tools, and vendors are expected to suit this to their needs and specifications.

8 Some implementation details

The code is written mostly in `C++` with extensive use of the STL, compliant with the 2014 standard. Some `C99` code is used, namely the GNU Scientific Library (GSL), for the wrapper and for solving algebraic loops at initialization time. For Windows users this prevents the code from being built entirely in Microsoft Visual Studio, since VS does not support `C99`. One way around this problem is to build GSL using `gcc` under `MSYS2` or `Cygwin`. This has been done successfully for the 32-bit build, but the 64-bit build still has some tedious, if solvable, problems. We hope to augment the system using the Sundial suite and other numerical integrators. More on this in Sec. 10.

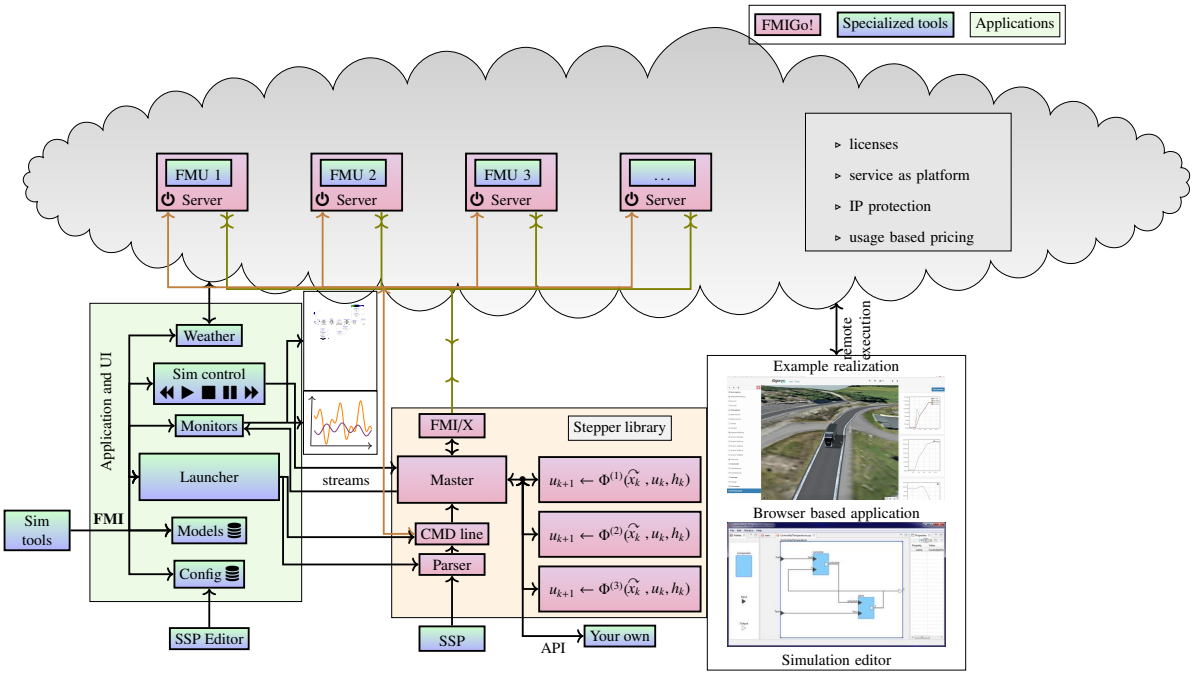

Figure 3: Global architecture

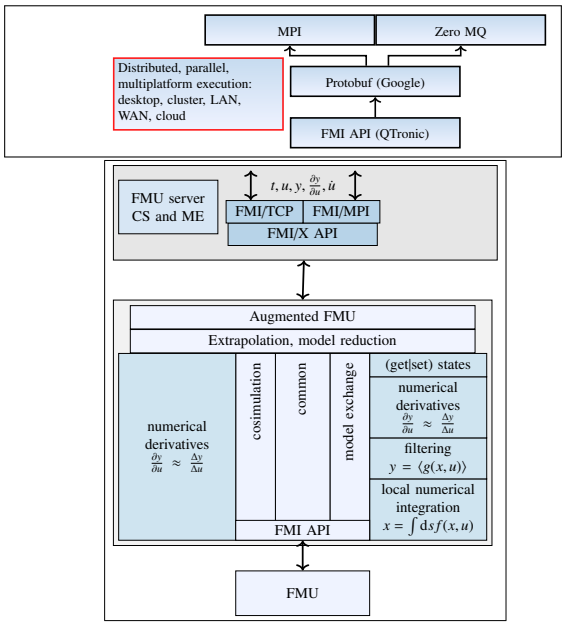

Figure 4: Abstraction layer and server

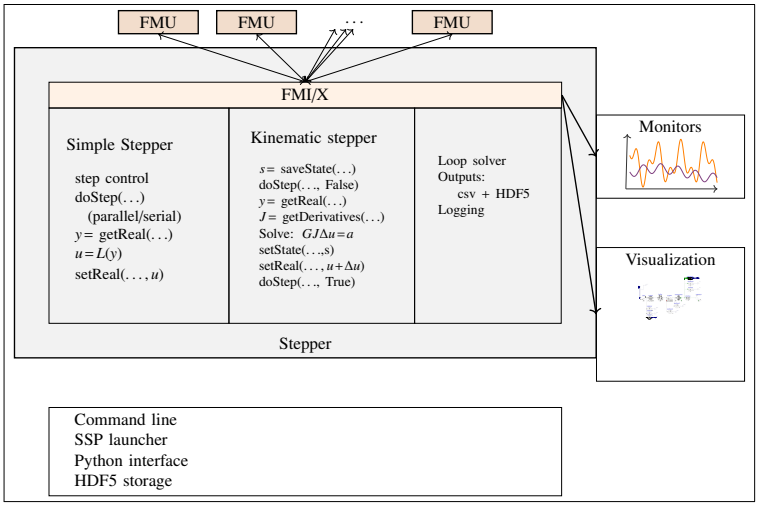

Figure 5: The client, master stepper

The build system uses `CMake` and is compatible with Windows, Mac OS and Linux platforms. We've also relied on continuous integration as available both on the GitLab platform and Jenkins. Binaries are built after each `git push` for various versions of Visual Studio as well as various Linux distributions, done via `docker`. After being built, the binaries are tested. This means we always know whether the system works on the desired platforms.

The kinematic stepper described in Sec. 14 requires the solution of linear systems of equations and this is done with UMFPACK [2].

Ancillary programs for implementing test cases, examples, data manipulation and plotting as well as SSP parsing, are written in `python3` using standard packages.

The native output format at this time is comma separated values in `ascii` format, as well as the MATLAB binary format version. This is less than optimal though we have started implementing HDF5 functionality with deep connections to the SSP. There are performance issues with HDF5 which aren't trivial to solve, requiring buffering, asynchronous writes and the likes. This is future work.

As previously said, we used Google's `protobuf` library to specify messages encapsulating the FMI API, and ZEROMQ for the TCP/IP implementation. The MPI implementation isn't entirely native at this point as it uses the `protobuf` messages instead of raw buffers for some elements of the FMI API. We've tested FMIGo! with different implementations of MPI, namely OPENMPI, MPICH, Microsoft's, as well as Intel's MPI implementations. All performed equally well though we observed that OPENMPI has bad performance when oversubscribing, i.e., when there are many more ranks than cores. We consistently observed latencies in the order of 120μs for a system with 8 nodes (1 master + 7 servers).

At the core of the solver is the reading and writing of signals from and to various FMUs. Given requirements for execution ordering where sequential and parallel parts are required, we opted for a directed graph structure allowing for nested combinations of parallel and sequential executions. This violates the FMI standard (which mandates that values between FMUs are exchanged at identical communication points), but is required in for example Gauss-Seidel stepping, and was also a requirement for a CAN emulator FMU at Scania. Thus we allow users to deviate from the standard if need be. The execution graph is statically constructed and decides in what order messages are pushed around, and when rendezvous are needed. Messages are batched such that as many as possible are sent before a rendezvous is required, which is typically for `fmi2GetXXX` calls following an `fmi2DoStep`. Depending on which FMUs have finished stepping, other FMUs are triggered.

The process of wrapping FMU is automated via CMAKE wherein rules are designed to take an existing FMU, produce an augmented `modelDescription.xml` file, and produce a new FMU with augmented capabilities such as directional derivatives and rollback.

Another component of FMIGo! is an abstracted interface to the Ordinary Differential Equations (ODE) Initial Value Problem (IVP) suite of GSL, `odeivp2`. This is the `cgsl` module. The point is that `odeivp2` is an extremely verbose API written in plain `C99` which makes it inflexible in and of its own and error prone. By encapsulating the library, we can easily define dynamical systems, modify the time integration method, and automatically filter the outputs or inputs with, for instance, what is described in Sec. 13. The module was designed so that new filters can be added easily. This could also be extended to support input extrapolations in a straightforward way. The same framework can be extended to use the SUNDIALS suite which supports sensitivity computations, providing better quality directional derivatives. Though we

did start this work, we have not completed it yet.

This `cgsl` module defines "poor man's" classes for `models` and `simulations` such that a simple example can be constructed in fewer than ten lines of code.

In order to construct our own simple test FMUs we modified the templates from Qtronics so that `cgsl` modules could be easily exported as FMUs. This is of limited use of course as it requires hand editing of XML files which is tedious and error prone. Yet, those simple FMUs can be used for quick tests. The main concept we used here is the construction of `C` structs derived from the `modelDescription.xml` file. An interface between an existing `C`-linkable module is readily obtained this way.

Then comes the PYTHON module `pygo` described in more details in Sec. 15. This defines `module` and `simulation` classes automating the generation of the command line necessary to launch FMIGo! Ancillary classes then convert the CSV data files to well structured HDF5 ones, and further functionality is provided to load these in `python` and plot data. Extensive support for variable aliasing is supported. This is described briefly in Sec. 15.

Documentation of the code and instructions on how to use the programs are found on gitlab `https://mimmi.math.umu.se/cosimulation/fmigo`

9 Performance measurements

No one likes performance degradation due to communication overhead and clearly, any distributed application will suffer with some of that. Also, the master-slave execution model of the FMI means necessarily that communication can become a bottleneck when there are very many FMUs.

In order to check how bad this overhead is, we ran several tests on the Abisko cluster at the High Performance Computing Center North (HPC2N) `www.hpc2n.umu.se`. We used the `gcc` compiler and Intel's MPI implementation, with trace instrumentation enabled or disabled depending on whether we wanted output for Intel's Trace Analyzer or not. The model being simulated was the typical mass-spring-damper chain (Fig. 10), where each middle mass is split in two and joined by a one-dimensional kinematic constraint. This coupling is described in further detail in Sec. 14. A system with N masses (before splitting) makes use of the following: $N-1$ FMUs, $N-2$ kinematic constraints and N MPI nodes. It involves solving an $N-2\times N-2$ sparse linear system of equations by the master at each step, and two signal rendezvous per step. Therefore the overhead per step grows roughly linear with the number of FMUs, 33-60 μs per FMU per step (Fig. 6).

For a view of where the time is spent for this minimal system see figures 7, 8 and 9. Fig. 7 provides an overview of MPI events in a system with 8 MPI nodes, 7 FMUs and 6 kinematic constraints. Initializing MPI takes some time, then 10,000 simulation steps are performed, then finally the system is torn down and the program ends. P0 is the master node, and P1 through P7 are the server nodes. A zoomed in view is provided in Fig. 8, showing communication during several steps. Two classes of rendezvous are clearly visible, with one class involving more computation inside the FMUs. For these trivial FMUs the calculations performed by the master node clearly dominates. In reality no one would distribute a trivial system such as this toy example, but it is useful for demonstrating the limits of our solution. Fig. 9 shows a fully zoomed-in view of a single step in the system, spanning roughly 180

μs. It is still possible to reduce this overhead somewhat, by avoiding `protobuf` calls for the messages required for the inner loop of the execution.

Optimizing `protobuf` overhead has already been done for the non-kinematic steppers, using simple C structs instead. For a test case involving densely connected, loosely coupled, trivial FMUs this optimization effort brought the execution speed for a system with 8 nodes on an Intel Core i7-860 CPU @ 2.80GHz up from 2.3 kHz to 8.8 kHz, nearly quadruple the speed! We don't expect quite so dramatic results for the kinematic solver, since we still have to do sparse matrix factorization at each step, but the potential savings in overhead are still likely to be substantial.

In summary, the execution speed of our solution depends on the number of FMUs, and the solver used. When using the kinematic solver one can expect speeds up to 3.3 kHz for small systems ($N = 6$) down to 21 Hz or less for very large systems ($N \geq 768$). Execution speeds for small systems not using the kinematic solver will be higher, at least 8.8 kHz for $N = 8$ if using a modern 8-thread CPU. Keep in mind that systems not using kinematic coupling must typically use shorter time steps, resulting in overall slower simulation speeds.

10 The CGSL module

The GNU Scientific Library (GSL) is a collection of modules useful to solve a variety of numerical problems, numerical time integration of ODEs. It is written in `C99`, which, as mentioned, poses a few issues with VisualStudio, though we have precompiled libraries which link without problem. Otherwise, `msys2` can be used. This module is meant to handle ME FMUs, but also, assist in the creation "by hand" of simple models. There are a few examples of these in the FMIGo! code.

Being a `C` library, the GSL is particularly verbose and it is difficult to make applications where one easily switch between integration and timestep control methods, and having default choices for instance.

The main benefit is the integration with the Qtronic FMU template which means that exporting FMUs from hand written code can be made simple. We are working on using the exact same `C` framework as an interface to `sundials` and other codes.

11 Cosimulation problem statement

There are cases where simulations are weakly coupled, such as, for instance, heat transfer and pipe flow simulations for district heating. One of the two needs mass flow from the other, the second needs temperature to adjust head loss parameters, or nothing at all. For this case, the two mathematical models are essentially orthogonal, and one provides boundary conditions for the other.

However, this is not the standard use-case. The coupled simulation of an engine and a vehicle is a case in point. The velocity of the engine and the drive shaft of the vehicle must coincide at all time. However, if the engine is simulated in isolation for a *communication step* or *sampling period* H, it has no load attached. Likewise, the vehicle might receive a constant torque for the same communication step which does not reflect reality.

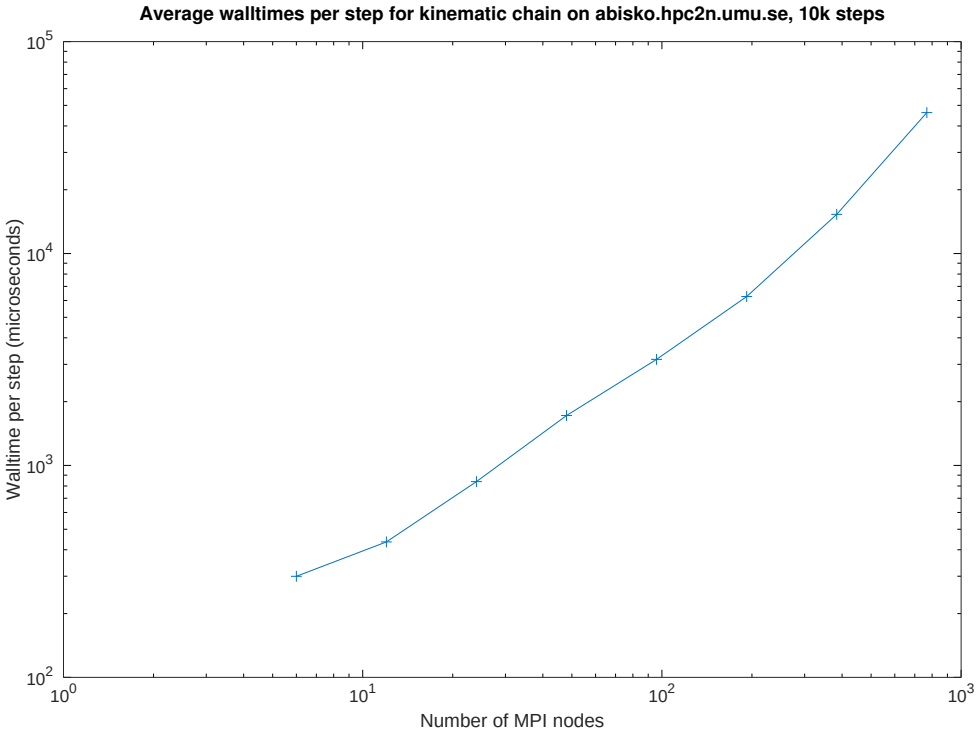

Figure 6: Average walltimes per step on abisko.hp2cn.umu.se for a kinematically coupled mass-spring-damper chain. Slope varies between 33 μs per FMU per step to 60 μs per FMU per step. Walltime includes setup and teardown, and 10,000 steps. No Trace Analyzer instrumentation.

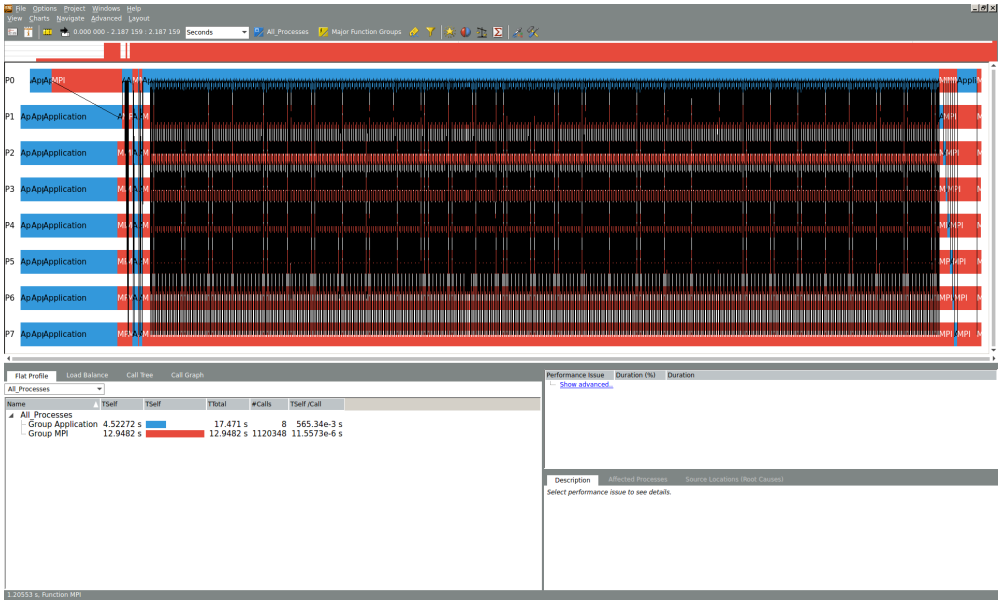

Figure 7: Trace Analyzer overview of kinematic chain with N=8 simulated on abisko.hp2cn.umu.se

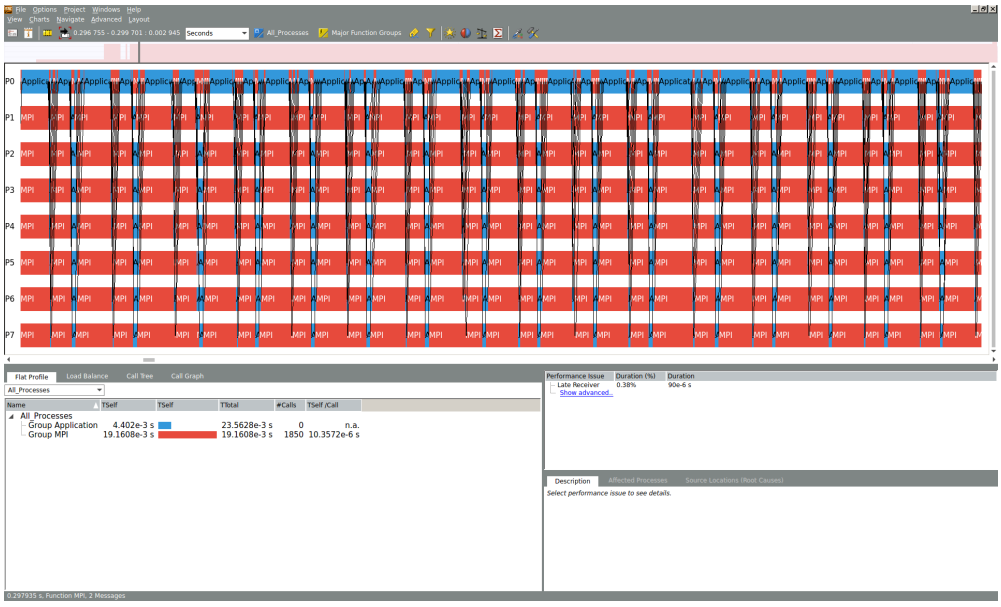

Figure 8: A closeup of several simulation steps of the trace shown in Fig. 7

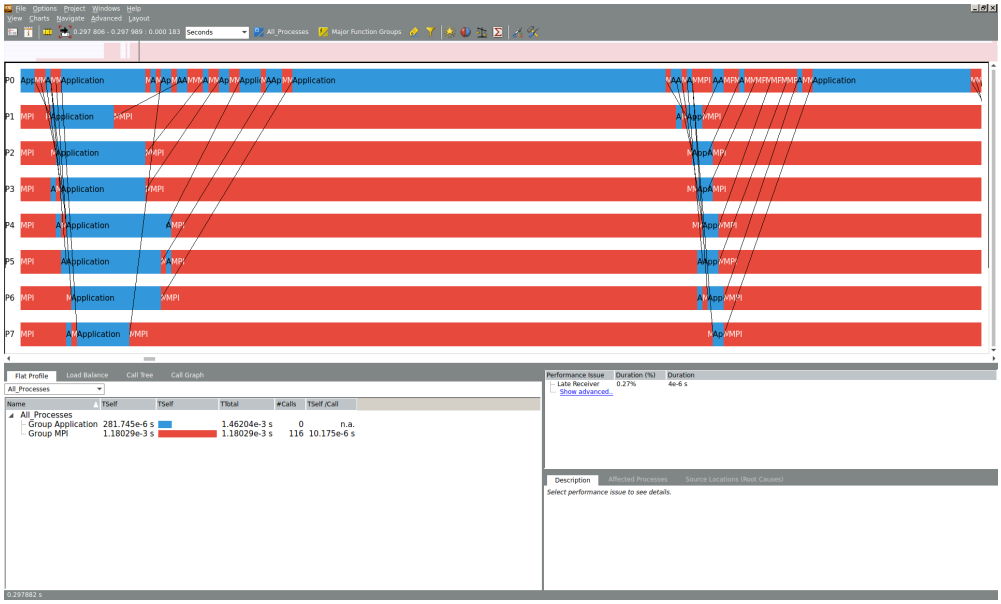

Figure 9: A closeup of a single simulation step of the trace shown in Fig. 7

The point is that the correct coupling is an algebraic condition representing a boundary condition. We focus on kinematic couplings in what follows such as, for instance, the condition that the angle of the output shaft of an engine must match that of the input shaft of a clutch.

Such couplings can be addressed with two broad classes of techniques. The first is based on force-velocity exchange which allows non-iterative techniques, not requiring any rollback or derivative functionality which is attractive in some cases.

The second is to enforce the kinematic condition directly. This requires rollback and directional derivatives from the FMUs which is not often available, as mentioned, a continuous ME FMU can be adapted to support that. Such techniques are necessarily "iterative" as corrections are needed to keep the kinematic conditions, and require a global solver. By "iterative" here we mean a method which requires several tentative step-forward attempts before reaching a satisfactory solution.

We consider these in the following two sections.

12 Force-velocity coupling

Here we lump the several variants of the idea into "force-velocity" coupling, in which one module communicates force to the other, and the latter then reports velocity back. Variants include sending both position and velocity, or sending positions (displacements) and velocities from both sides, etc.

For the force-velocity case, one module has a spring damper "adapter" to convert an input

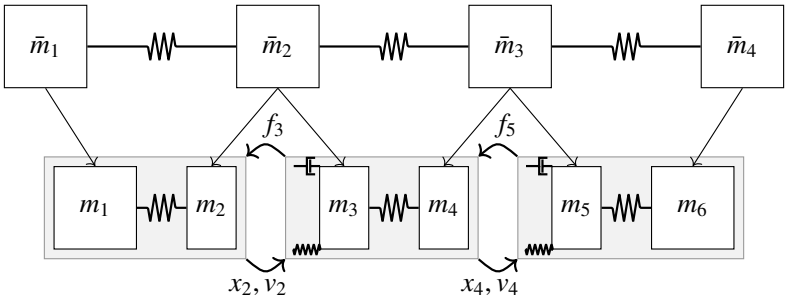

Figure 10: Splitting a chain into modules

velocity into a force, for instance. The fundamental problem here is that without a global solver, we cannot compute interaction forces. But each module which outputs, say, a velocity, must receive a counter-force in some way. If we consider two systems which should be simulated as

For the case at hand, an input shaft would be connected to the spring damper. At each communication step, this module receives a velocity which is then used to drive on end of the spring. The other module just sends an output force, but receives also the counter force from the other module which is computed from the spring compression.

There are several issues with this. First off, the adapter spring-damper is not part of the original model and introduces undesired frequencies into the system. The trick is to force those to be high enough and dissipation to be quick and small enough as to not interfere with the dynamics under study. A good rule is that this should be 30 times higher than the "natural" dynamics of the module as shown below. However, this increases the stiffness of the system and, necessarily, cuts the time steps by a factor of 30, unless good stiff integrators are used. Though that can bring other problems such as numerical damping. The next problem is that with such high frequency output, the coupled module gets noisy inputs. Since for the "non-iterative" case – only one sweep through modules which advance by one communication step each but transfer "future" data to others – the global stepper is nothing else than forward Euler, this noise brings the time step further down. Overall, one often gets three orders of magnitude penalty: 30 for each module with a spring-damper adapter, 30 more at the global stage.

This is a serious impediment to modular simulations, and begs for better methods which can advance with large step. As these are iterative in nature, meaning that they require rollback, there is significant motivation to provide wrapper FMUs which implement these essential features.

13 Energy preserving filters

Coupling between subsystems can be achieved by introducing artificial forces in system i of the form

$$f^{(i,j)}(t) = -K(x^{(i)}(t) - x_k^{(j)}) - d(v^{(i)}(t) - v_k^{(j)}), \text{ for } t \in [t_k, t_{k+1}], \tag{17}$$

and where $x_k^{(j)}$ and $v_k^{(j)}$ is the last known value from system j. In practical terms, it is often better to use

$$\begin{aligned} f^{(i,j)} &= -K(\delta x) - d(v^{(i)}(t) - v_k^{(j)}), \text{ with} \\ \delta\dot{x} &= v^{(i)} - v_k^{(j)}, \text{ and } \delta x(t_k) = 0. \end{aligned} \tag{18}$$

With either form, the oscillations produced by this artificial springs are expected to high lest this interferes with the dynamics of the system. The argument here of course is that as $K \to \infty$ and the step $H \to 0$, the two subsystems are in sync.

Now, system j injects kinematic information into i, and in return, system i reports $-f_{k+1}^{(i,j)}$ at the end of the communication step. But the high frequency oscillations mean that, unless the system is heavily damped, the output value can vary quickly which is bad. Likewise, if the inputs to system i vary rapidly, then it is excited very strongly.

An idea is to filter this [3] by introducing new variables in the equations of motion of system i, namely

$$\dot{z}^{(i,j)} = f^{(i,j)}(t), \quad z^{(i,j)}(0) = 0, \quad t \in [t_k, t_{k+1}]. \tag{19}$$

We write $z_k^{(i,j)}$ for the value at the end of the integration interval. We can then report an average such as

$$\langle f_{k+1}^{(i,j)} \rangle = \frac{1}{2H}\left(z_{k+1}^{(i,j)} + z_k^{(i,j)}\right). \tag{20}$$

If we assume a signal of the form $f = \sin \omega t$, then we have

$$\begin{aligned} z(t) &= \frac{1}{\omega}(\cos \omega t_k - \cos \omega t), t \in [t_k, t_{k+1}] \\ z_{k+1} &= \frac{1}{\omega}(\cos \omega k H - \cos \omega (k+1)H), \end{aligned} \tag{21}$$

so that, after trigonometric manipulations

$$\langle f_{k+1} \rangle = \left(\frac{\sin \omega H}{\omega H}\right) \sin \omega k H = \left(\frac{\sin \omega H}{\omega H}\right) \sin\left(\omega(k+1)H - \omega H\right). \tag{22}$$

Abusing notation, the continuous time representation is

$$\langle f \rangle(t) = \left(\frac{\sin \omega H}{\omega H}\right) \sin\left(\omega t - \omega H\right), \tag{23}$$

making the phase shift obvious.

As for the amplitude, we rewrite this in terms of the period $T = 2\pi/\omega$ and so the filtered value is then suppressed by

$$\langle f_{k+1} \rangle = \left(\frac{\sin 2\pi(H/T)}{2\pi(H/T)}\right) \sin 2\pi k (H/T). \tag{24}$$

From this it is clear that when the sampling period H is half that of the oscillator, the signal vanishes completely and the same occurs whenever $2H/T$ is an integer. The amplitude of the filtered signal also decreases as T/H meaning that for larger sampling period H, the lower

the amplitude. Viewed another way, this can describe perfectly stable simulation with entirely inaccurate results.

The phase shift corresponds to a lag which isn't surprising but causes some issues, especially when long chains of coupled modules are considered.

Now, the problem here is that the filters must be integrated along with the rest of the system's dynamics. Therefore, these variables must be added to the original model, and therefore, cannot be used at all when using someone else's cosimulation FMUs.

However, FMIGo! provides a wrapper for ME FMUs which then become either augmented ME FMUs, or CS FMUs. The wrapper automatically adds these filtered variables without any intervention from the user. The transformation to CS FMUs should not be necessary with a fully functional hybrid stepper, even though the filtered variables would be useful when exchanging signals from the continuous to the discrete time models.

14 Kinematic coupling

A natural way to couple mechanical systems is to "cut" bodies which connect two distinct parts, such as a shaft between an engine and a clutch, or an axle connecting a wheel to a chassis. Because of this, the coupling between two subsystems is often a kinematic one i.e., the angle of the output shaft from the engine must match that of the input shaft of the clutch. Such kinematic couplings are algebraic conditions turning the system into a Differential Algebraic Equation (DAE).

We use methods from multibody dynamics to construct a solver which computes coupling torques to maintain such conditions which we call constraints henceforth. The numerical time integration method we use is based on Spook [9] which is a variational integrator and is a variant of Shake [7] which introduces symmetry in time for constraint satisfaction. Here we preserve an average of the constraints in order to do a symmetric linearization. Symmetric methods have proven very effective for other DAE systems as well [5].

Briefly, we consider a mechanical system with mass M, Cartesian coordinates x velocities $v = \dot{x}$ and forces forces f, and displacement functions $g(x)$ which have Jacobians $G = \partial g/\partial x$. We want to constrain the system to the manifold $\mathcal{M} = \{x \mid g(x) = 0\}$. This requires forces normal to the manifold $G^T\lambda$ where λ are Lagrange multipliers. We neglect details related to spatial systems where generalized coordinates are needed to represent rotations. This means that we also neglect gyroscopic forces for simplicity and consider M to be constant.

At the numerical level we use,

$$v_k \approx \frac{1}{h}\left(x_k - x_{k-1}\right), \text{ meaning that } x_{k+1} = x_k + hv_{k+1}, \tag{25}$$

where k is the discrete time. This is akin to the leapfrog method [7] and is significantly different from the standard forward Euler method.

Instead of imposing the constraints $g(x_k) = 0$ directly at each step k, we instead impose

$$\frac{1}{4}(g_{k+1} + 2g_k + g_{k-1}) + \tau G_{k+1}v_{k+1} + \frac{1}{h}\epsilon(h\lambda) = 0. \tag{26}$$

In practice, we solve for $h\lambda$ directly and by abuse of notation, we alias $h\lambda$ to λ. The τ term corresponds to damping of the averaged constraint violation, and the ϵ term introduces relaxation. This is needed for the case where G is rank deficient.

After linearization we get the one stage stepping

$$\begin{bmatrix} M & G_k^T \\ G_k & \tilde{\epsilon} \end{bmatrix} \begin{bmatrix} v_{k+1} \\ \lambda \end{bmatrix} = \begin{bmatrix} Mv_k + hf_k \\ -\frac{4\gamma}{h} g_k + \gamma G_k v_k \end{bmatrix} \tag{27}$$
$$x_{k+1} = x_k + hv_{k+1},$$

where

$$\tilde{\epsilon} = \gamma \frac{\epsilon}{4h^2}, \text{ and } \gamma = \frac{1}{1 + 4\tau/h}. \tag{28}$$

This scheme is stable for $\tau > 0$, and maintains constraint satisfaction within $O(h^2)$ as shown previously [8]. In practice, we choose $\tau = 2h$ which is optimal in terms of the time it takes to relax the average constraint violation. Also, if there is no elasticity in the system but constraint degeneracy, we choose $\gamma\epsilon = 10^{-8}4h^2$ which keeps the condition number of the matrix on the left hand side of Eqn. (27) near 10^{-8}.

An equivalent representation of the stepping scheme uses the Schur complement and reads

$$\left[G_k M^{-1} G_k^T + \tilde{\epsilon}\right] \lambda = -\frac{4\gamma}{h} g_k + \gamma G_k v_k - G_k v_k \left[v_k + hM^{-1} f_k\right]. \tag{29}$$

Note that the terms in the last square brackets to the right correspond to the velocity at time $k + 1$ in the absence of constraint forces, something we use later.

In this formulation, the matrix on the left hand side is recognized as the mobility of the multibody system, i.e., numerically

$$G_k M^{-1} G_k^T \approx \frac{\partial G_k (v_{k+1} - v_k)}{\partial h f_k}. \tag{30}$$

This is therefore the change in v_{k+1}. The pertinence of this for the cosimulation case will be clear shortly.

Let's now consider two systems with output and input signals $y^{(i)}, i = 1, 2$ and $u^{(i)}, i = 1, 2$, respectively, such that $y^{(i)}$ contains positions to be constrained and corresponding velocities $x^{(i)} = J^{(i)} y^{(i)}$ and $v^{(i)} = K^{(i)} y^{(i)}$, and the inputs $u^{(i)}$ allow to apply a force $f^{(i)} = G^{(i)^T} \lambda^{(i)^T}$. Here we would like to impose the constraints

$$g^{(l)}(x) = 0, \tag{31}$$

and the functions $g^{(l)}$ is implemented in the global stepper, as well as Jacobians $G^{(l,i)}$.

For the cosimulation case we only have access to the time translation operators

$$x^{(i)} = \Phi_H^{(i)}(u_k^{(i)}) \text{ and } v^{(i)} = \Psi_H^{(i)}(u_k^{(i)}), \tag{32}$$

meaning that we do not know $M^{(i)}$. We assume now that we have access to the directional derivatives

$$\frac{\partial \Psi_H^{(i)}}{\partial u_k^{(i)}} \tag{33}$$

as finite differences at least. To construct a linearization based on discrete time translation operator, we first assume that

$$x_{k+1}^{(i)} \approx x_k^{(i)} + Hv_{k+1}^{(i)}, \tag{34}$$

as done in Eqn. (25). The goal now is to recover Eqn. (29), but the problem is that we do not have access to the equivalent of the terms

$$-G\left[v_k + M^{-1} f_k\right]. \tag{35}$$

However, assuming that the signals $u_k^{(i)}$ are purely additive for simplification, then the first guess for the velocity $\bar{v}^{(i)}k + 1$ is

$$\bar{v}_{k+1}^{(i)} = \Psi_H^{(i)}(u_k^{(i)}) \approx v_k^{(i)} + H\left(M^{(i)}\right)^{-1}(f_k^{(i)} + u_k^{(i)}), \tag{36}$$

meaning that

$$v_{k+1}^{(i)} \approx \bar{v}_i^{(k+1)} + \delta\tilde{u}^{(i)} \tag{37}$$

where $\tilde{u}$ is the part of the inputs which applies a force. If we now try to construct the linearization in Eqn. (30), we need the approximation

$$\begin{aligned} g^{(l)}(x_{k+1}) &= g^{(l)}(x_k) + \sum_i G_k^{(l,i)} x_{k+1}^{(i)} = g^{(l)}(x_k) + H\sum_i G_k^{(l,i)}\Psi_H^{(i)}(u_k + \delta u) \\ &= g(x_k) + H\sum_i G_k^{(l,i)}\bar{v}_k^{(i)} + H\left[\sum_i G_k^{(l,i)}\frac{\partial\Psi_H^{(i)}}{\partial u_k^{(l)}}\right]\delta u^{(l)} \end{aligned} \tag{38}$$

Note that δu is a vector, with one element per constraint, and that for two bodies i, j connected via $g^{(l)}$, we have $\delta u^{(l,i)} = -\delta^{(l,j)} = \delta u^{(l)}$. When all terms are considered, the new form of Eqn. (29) is

$$\left[\sum_i G_k^{(i)}\frac{\partial\Psi_H^{(i)}}{\partial u_k^{(i)}} + \tilde{\epsilon}\right]\delta u = -\frac{4\gamma}{h}g_k + G(\gamma v_k - \bar{v}_k). \tag{39}$$

The l superscripts have been dropped so Eqn. (39) is to be understood vectorially, meaning that δu is not a vector with components for each constraint $l = 1, 2, \ldots$.

This is illustrated in Fig. 11 in the context of rotational elements.

As experimental verification, we present two examples. The first is based on modules containing two point masses connected by a spring and damper. The second, more complicated, is a truck model with a simulated driver, an engine, a gearbox, an axle, and a full 3D multibody representation of the wheels, chassis, and trailer. Note that the gearbox, axle and multibody modules contain discontinuities which are not visible to the global stepper.

For the spring-damper model, we used different masses with ratios of up to 1000:1, though that only worked with kinematic coupling. Indeed, large mass ratios often cause big problems for force-velocity couplings as they lead to less stability. We tried a variety of integrators for the modules and found, to our surprise, that we could reach a fourth order convergence in the constraint violation with the Dormand Prince pair of order 8. Our experiment was designed so the largest communication step was in the range of $1/5$ of the smallest period in the system, which is what a good, high accuracy integrator would use as time step when solving such a

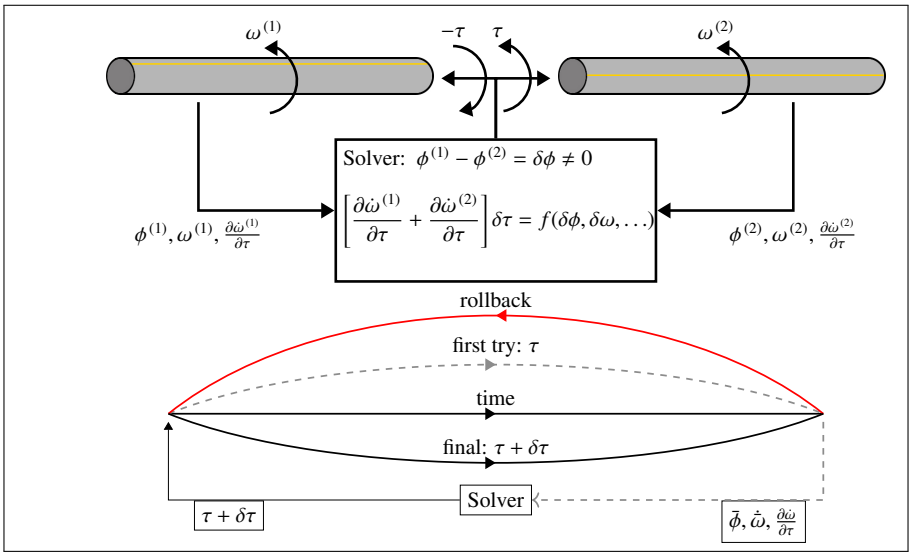

Figure 11: Schematics of the solver's iteration procedure

linear system. The x axis in Fig. 12 corresponds to the amount of work done, and represented as *number of steps per period*. Unusual as this might seem, this is the natural, dimensionless way to represent the time step magnitude. Of course, it might not be possible to measure the shortest period for a complicated system, yet time scales can always be estimated.

15 Yet another support library

The complexity of running FMI based simulations is considerably larger than that of, say, running a Simulink model. And though a good SSP editor is the only solution for large systems, simple scripts are needed to test simulations and analyze data. These should be cross platform as much as possible.

We developed a `python` support library to assemble simple simulations which abstracts away the intricacies of launching FMIGo! and reformatting the output data in a manageable way with HDF5. We declared two main classes: a module which encapsulates a given FMU, and a simulation which is a collection of these along with couplings etc. The module is called `fmisims` and is located in the python support script directory. Every effort was taken to make this portable between Linux, Windows and Mac OS X. We used `python3` and made no effort towards `python2` compatibility.

For instance, a given truck FMU would then be augmented as follows:

```
class truck(module):
    """ The AGX truck """
    def __init__(self):
        module.__init__(self,
            filename="algoryx/TruckStrong.fmu", name="AGX_truck",
                    connectors={"drive_shaft" :
                                ["phi_drive_shaft",
                                 "shaftConnector.output.angularVelocity.z",
```

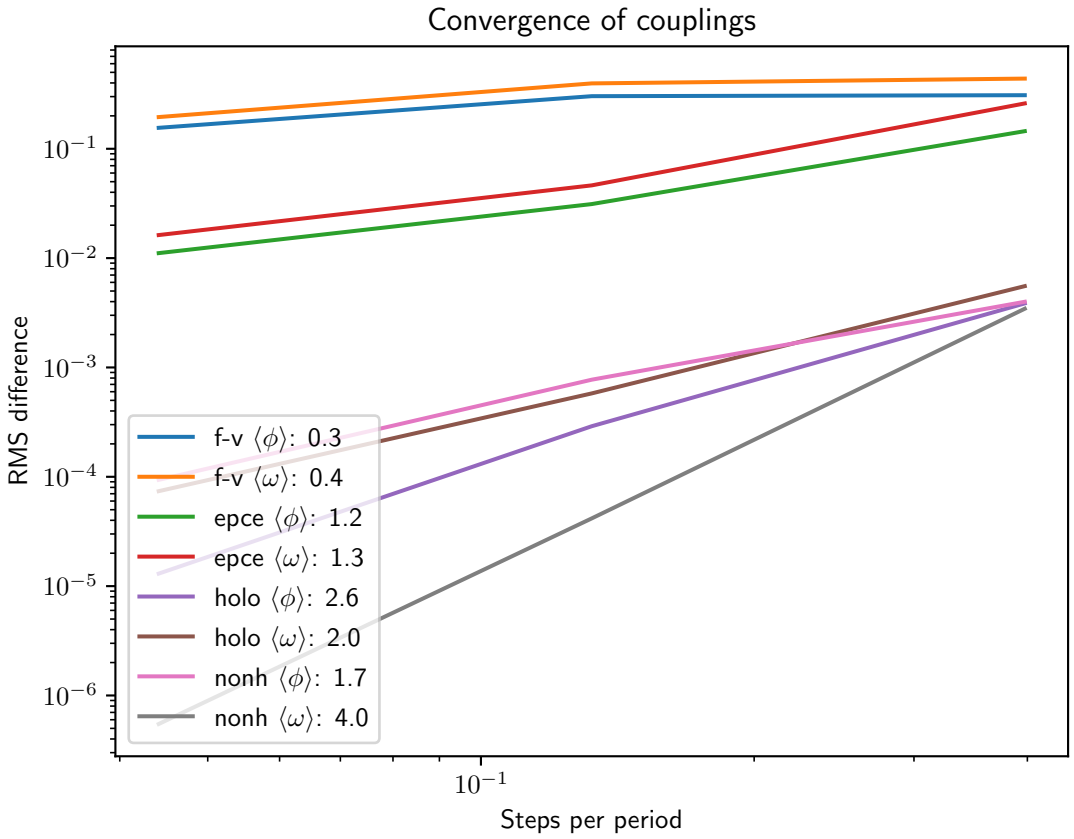

Figure 12: Convergence of the kinematic stepper

```
                    "shaftConnector.output.angularAcceleration.z",
                    "shaftConnector.input.torqueAccumulator.z"]},
            outputs={"velocity" : "truck_v_vehicle_mps_R",
                     "phi_drive_shaft" : "phi_drive_shaft",
                     "w_drive_shaft" : "shaftConnector.output.angularVelocity.z",
                     "input_torque" : "shaftConnector.output.torqueAccumulator.z",
                     "w_drive_shaft_smooth" : "w_shaft_smoothed"
            },
            inputs={"gear" : "drvtn_s_gear_no_R",
                    "clutch_torque" : "drvtn_tq_clutch_nm_R",
                    "input_torque" : "shaftConnector.input.torqueAccumulator.z"},
            description="AGX truck model"

    )
```

What "connectors" and "outputs" do is to create aliases which make it easier to interchange equivalent FMUs. Assuming a similar FMU for, say, a driveline, the full simulation would then look like

```
class agx_agx(simulation):
    def __init__(self):
        simulation.__init__(self, {"driveline" : agxdl() , "truck" : truck()},
                            name = "truck",
                            variant ="agx_agx_kinematic",
                            couplings=(("driveline", "drive_shaft", "truck", "drive_shaft"),),
                            signals=( ("truck", "velocity", "driveline", "velocity"), ),
                            description="AGX truck and driveline with kinematic coupling")
```

The "couplings" tuples establish kinematic connections, while "signals" are just values which are pushed around. Running the simulation is then performed using:

```
s = agx_agx()
s.simulate(2, 0.01, mode="mpi", holonomic=holo, datafile="data")
```

This will produce both a comma separated value `data.csv` and a HDF5 `data.h5` files containing the simulation data. The HDF5 file contains ample annotations as well as alias tables for the case where we compare different equivalent simulations which use different FMUs with different names for the corresponding variables. The annotations include the communication step, an identifier for the models used, the operating system, some identification of the specific computer, the wall time taken for the simulation, and much more.

A number of example simulations are provided in the module `libmodels` which provides, in particular, class definitions for the `umit-fmus` modules which are simple, hand coded FMUs good for testing ideas.

HDF5 files are well suited to contain many large datasets and therefore, the framework is designed to accumulate results from different simulations with different parameters and different equivalent modules in a single file. From the data analysis point of view, this presents a few difficulties as one might be interested in only a subset of the data in the file. This should be assembled in lists so that plots can be constructed using data from different, similar simulations.

The module `h5load` provides two main functionalities. First an object `sim_data` which includes all the metata data found in the HDF5 file as a dictionary, and a `pytables` table containing the data. Second is a function `package_simulations` which finds simulations in the HDF5 files which meet a set of criterion. This is done with a support class `match_value` which is a polymorphic test function which traverses the file hierarchy and tries to match presence or values of HDF5 attributes. The `match_value` class is design to accept any type which can be held by an HDF5 attribute written from the `pytables` module with the `mygroup._f_setattr` function applied to a HDF5 group `mygroup`. Note that this function preserves the type of the attribute which can be a string, integer, floating point number, dictionary and any other python

type. For instance, assuming a series of simulations which used either holonomic or nonholonomic couplings on a truck model which has been defined using two different sets of FMUs, the analysis would involve extracting the different variants. The listing below will look for simulations that ran for approximately four seconds with any communication step size, and choose only those runs involving the specific model `dym_point_mass_axle_kinematic` with kinematic coupling using nonholonomic coupling. The result is then a sorted list in descending order of communication step, and each element in the list is a `sim_data` object.

```
dym_nonh = sorted( package_simulations(mlocate_nodes(f.root, "Group", conds = [
    match_value(range(low=3.9, up=4.1), "tend"),
    match_value("truck", "name"),
    match_value("dym_point_mass_axle_kinematic", "variant"),
    match_value("kinematic", "coupling"),
    match_value("nonholonomic", "coupling_type") ] ), f, close_file=False),
                key=lambda y: float(y.get_attr("comm_step")), reverse=True )
```

To use this for, say, a convergence analysis, one can use the listing below. This uses support functions for computing the global error in constraint satisfaction, computes some sort of average of global error for the different steps, and then produces a loglog plot after formatting keys properly. The `global_err` function illustrates how the data is accessed in the `sim_data` class which has the form

```
sim["fmu_name"]["variable"]
```

where `sim["fmu_name"]` returns a `numpy` structured array such that columns of data can be accessed by name.

```
def global_err(m, which, how="rms"):
    ds= "%s_drive_shaft" %which

    x = abs(np.column_stack([m["engine"]["%s_flywheel" %which]-m["gearbox"]["%s_input_shaft" %which],
           m["gearbox"]["%s_output_shaft" %which]-m["axle"]["%s_prop_shaft" %which],
                          m["axle"]["%s_drive_shaft" %which]-m["truck"][ds] ]))
    if how == "rms":
        return sqrt(x[:, 0]*x[:, 0] + x[:, 1]*x[:, 1] + x[:, 2]*x[:, 2]) / 3.0  + 1e-16
    elif how in [0,1,2]:
        return x[:,how]+eps
    else:
        return x+eps


dhdphi= []
dhH = []
for i in  dym_nonh:
    dphi = global_err(i, "phi", how=0)
    dhdphi += [ mean(dphi) ]
    dhH += [ float(i.get_attr("comm_step")) ]

axc = [ loglog(dhH, dhdphi)]
xlabel(r"Communication step $H$ [$\mu$s]")
ylabel(r"$\log_{10}( \langle \|\delta \phi\|\rangle $)")

title("Global error convergence, %s" %(sim[0].get_attr("variant")))
legend("$\phi$ nonholonomic" + " %1.1f" % polyfit(log10(dhH), log10(dhdphi), 1)[0] )
```

This script is part of the available examples which use the `umit-fmus` simple modules.

Data in the HDF5 file can be explored with commonly available tools such as `hdf-java` or `vitables`, both of which are cross platform. Other tools can load HDF5 datasets of course. However, the `h5load` module provides ample functionality for producing plots using `matplotlib` which are of sufficiently high quality in most cases, and easily integrated with LaTeX.

Conclusion

FMIGo! provides a solid foundation for manipulating FMUs. Being distributed and protocol based, it "plays nice" with other tools which often have XML type formats which are easy to implement. Much easier than plugins for instance.

Compliance with the emerging SSP standard should make it attractive to those who have tools for generating SSP files already, and avoid going through an overly simple GUI editor. PYTHON modules also help generating quick examples programmatically which is less error prone than point-and-click.

As for future work, time constraints mean that we have not managed to construct a fully functional hybrid stepper yet, and that wrapper FMUs have only limited, though very useful, functionality yet.

The EPCE implementation is performing as expected and with automatic adapters in the wrapper FMUs, should be useful to those who have no intention of modifying their models to fulfill requirements imposed by model integration. This needs much more work of course since there are many variations of filtering algorithm. Yet this algorithm can deliver good results for moderate communication step H and is therefore a good alternative for those who can only perform non-iterative coupling with their modules.

The kinematic stepper offers second order convergence and should be preferred when algebraic constraints are the natural, painless way, to define connections between modules. Note however that this only apply to second order systems with DAEs of index 3 as found in multibody systems dynamics in descriptor form. Future work includes checking whether or not this can be adapted to other types of DAEs.

Full connection between SSP files and data files is sorely missing at this time, so is a suitable binary format to store the runtime data. Previous experience with burst write to HDF5 [10] shows that this is possible but time constraints forced us to delay this work.

Federations or hierarchies as described previously [4] as well as peer-to-peer communication implemented by other libraries [1, 11, 13] isn't implemented but given the ambiguity of the standard with respect to this kind of execution paradigm, shouldn't matter too much.

Implementation of error control based on Richardson extrapolation [12] isn't implemented either by contrast to a competing effort [4]. This too is sorely missing and part of future work.

The installation of FMIGo! from sources is fairly easy and extensive instructions are provided Integration (CI) as provided by JENKINS and GITLAB meaning that the time it takes to get running is significant.

A full fledged end user environment was realized by ALGORYX already at time of writing. We expect others, and we hope for academic collaborations for the numerical methods, and some headway was made in that direction already. Connection with the Horizon2020 projects ENTOC and SPEAR, are succeeding.

We have yet to recruit more partners to make this project radiate beyond the original partners but we are still confident that we can expand on the number of numerical methods in the main stepper, and provide basic "do one thing and do it right" type tools to make FMIGo! an attractive option to try new thing, without secrecy or copyright constraints.

Of course, this project needs a continuation given the list of open issues mentioned in the present and we are of course looking for ways to finance the next phase which would consolidate the core functionality. Nevertheless, we will keep following the "do one thing and do

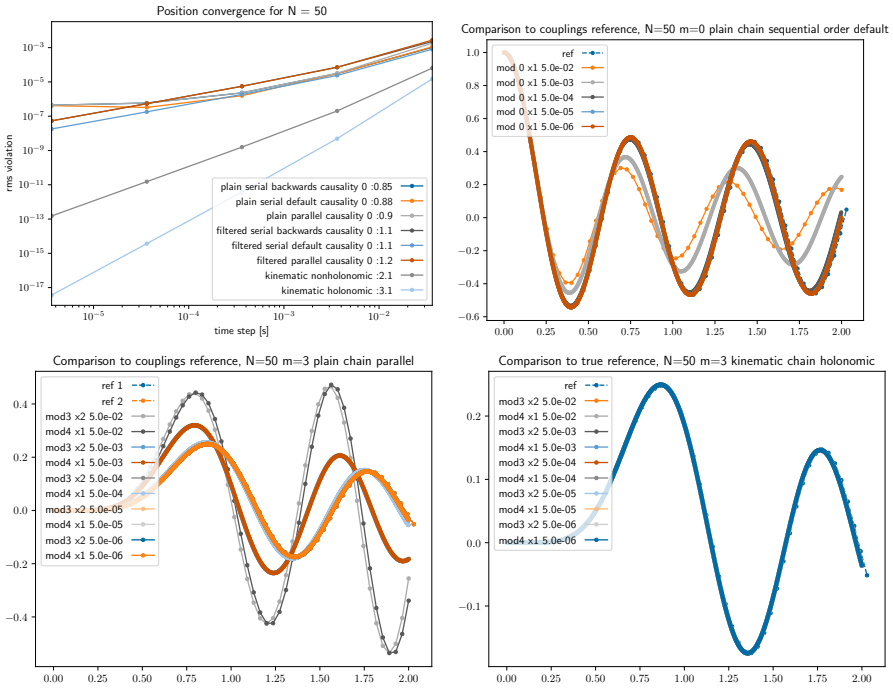

Figure 13: Convergence on spring damper chains

it right" philosophy meaning that FMIGo! will never be a complete end user tool, but only a reliable work horse.

Acknowledgments

The early development of the FMIGo! was supported via the Vinnova grant Simovate, No 2012-01235, with contributions from Stefan Hedman between 2012 and 2014.

The second phase leading to a usable piece of software was financed by the Swedish Energy Agency under the Strategic Vehicle Research and Innovation (FFI) in collaboration with Scania, Algoryx Simulation, Volvo Cars, and Modelon, during 2016 and 2018.

The program FFI is a partnership between the Swedish government and the automotive industry aiming to jointly fund research, innovation and development activities focusing on the areas of climate and environment, and security.

In the first phase, Stefan wrote the code and read Claude's mind.

For the second stage, Tomas wrote almost all the new code, came up with bright ideas, and criticized everything to dust, the Scania team provided models, objectives, and much patience, the Algoryx team realized the full fledged cloud and web based environment, models, and much patience as well. Edo Drenth from Volvo cars provided the energy preserving

filters and Mats Johansson from UMIT provided technical and valuable administrative support. Claude did most of the math as well as the `cgsl` and `pygo` modules and much of the software architecture.

Finally, where would we be without the geniuses answering questions on stackexchange?

References

[1] David Broman, Christopher Brooks, Edward A. Lee, and Thierry Stephane Nouidui. org.ptolemy.fmi. `http://tinyurl.com/ptolemy-fmi`, 2013.

[2] Timothy A. Davis. Algorithm 832: UMFPACK — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, June 2004.

[3] Edo Drenth. Robust co-simulation methodology of physical systems. In *9th Graz Symposium Virtual Vehicle*, May 2016.

[4] Virginie Galtier, Stephane Vialle, Cherifa Dad, Jean-Philippe Tavella, Jean-Philippe Lam-Yee-Mui, and Gilles Plessis. FMI-based distributed multi-simulation with DACOSIM. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS '15, pages 39–46, San Diego, CA, USA, 2015. Society for Computer Simulation International. ISBN 978-1-5108-0105-9. URL `http://dl.acm.org/citation.cfm?id=2872965.2872971`.

[5] E. Hairer. Symmetric projection methods for differential equations on manifolds. *BIT Numerical Mathematics*, 40:726–734, 2000. ISSN 0006-3835. doi: 10.1023/A:1022344502818.

[6] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Non-stiff Problems*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second revised edition, 1991. ISBN 3-540-56670-8.

[7] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2001.

[8] Claude Lacoursière. *Ghosts and Machines: Regularized Variational Methods for Interactive Simulations of Multibodies with Dry Frictional Contacts*. PhD thesis, Dept. of Computing Science, Umeå University, SE-901 87, Umeå, Sweden, June 2007.

[9] Claude Lacoursière and Mattias Linde. Spook: a variational time-stepping scheme for rigid multibody systems subject to dry frictional contacts. Technical Report UMINF 11.09, Dept. of Computing Science, Umeà University, September 2011. URL `http://www8.cs.umu.se/research/uminf/index.cgi?year=2011\&number=9`.

[10] Claude Lacoursière and Sjöström. A non-smooth event-driven, accurate, adaptive time stepper for simulating switching electronic circuits. Technical Report UMINF 16.15, Dept. of Computing Science, Umeå University, December 2014.

[11] Edward A. Lee. Constructive models of discrete and continuous physical phenomena. Technical Report UCB/EECS-2014-15, EECS Department, University of California, Berkeley, February 2014. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-15.html.

[12] Arnold Martin, Clauss Christoph, and Schierz Tom. Error analysis and error estimates for co-simulation in FMI for Model Exchange and co-simulation v2.0. *Archives of Mechanical Engineering*, 60:75–94, March 2013. doi: 10.2478/meceng-2013-0005. URL //www.degruyter.com/view/j/meceng.2013.60.issue-1/meceng-2013-0005/meceng-2013-0005.xml. 1.

[13] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL http://ptolemy.org/books/Systems.

[14] QTronic. QTronic FMI SDK. http://www.qtronic.de/en/fmusdk.html, 2017.

[15] Tom Schierz, Martin Arnold, and Cristoph Clauss. Co-simulation with communication step size control in an fmi compatible master algorithm. In *Proceedings of the 9th International MODELICA Conference*, September 2012.

[16] Bernard P. Zeigler, Herbert Praehofer, and Tag G. Kim. *Theory of Modeling and Simulation, Second Edition*. Academic Press, 2nd edition, January 2000. ISBN 0127784551.