Experience with Industrial In-House Application of FMI

Kilian Link[1] Leo Gall[2] Monika Mühlbauer[3] Stephanie Gallardo-Yances[4]

[1,3,4]Siemens AG, Germany, {kilian.link, monika.muehlbauer, stephanie.gallardo}@siemens.com

[2]LTX Simulation GmbH, Germany, leo.gall@ltx.de

Abstract

This paper discusses FMI usage in an in-house simulation tool landscape where it helps to open doors between different tools. However, limitations due to missing physical connectors or missing model structure are faced and are described with the help of use cases.

Information hiding in FMI can turn out obstructive in in-house applications. Experiences from implementing FMI support in in-house simulation tools are shared.

Keywords: FMI in in-house simulators, FMI2.0, physical connectors, structured parameters, co-simulation, model exchange, NMPC, control test

1 Introduction

The industrial application of Functional Mock-Up Interface (FMI) has already been discussed several times, e.g. (Bertsch et al., 2014). The goal of this paper is to add another aspect to this discussion. Our focus lies on the in-house applicability of FMI in coupling different tools and propriety models.

For several years, we are developing side-by-side two different simulation tools for power plant systems. An in-house Modelica library, called SiemensPower, and a C++ based in-house tool for a similar purpose named Dynaplant. The reason for developing both simulation tools lies in their different strengths and weaknesses. Dynaplant scores with its high performance and numerical robustness that allows us the investigation of fluid systems with a hundred thousand dynamic states or more. It is fully integrated into the in-house tool chain and allows an automatic model setup from the design software KRAWAL® for steady-state heat balances. The GUI is optimized for the purpose of modeling large fluid systems and the plant model looks very similar to the familiar KRAWAL® model. The drawback of Dynaplant is the relatively high effort when developing new component models.

The main driver for starting a Modelica library was the modeling flexibility and transparency to the user. Naturally, a Modelica library requires less effort in, developing and maintaining of models. However, the performance and numerical robustness is worse.

Once the need for two simulation environments is accepted, it stands to reason to combine them in order to leverage the benefits of both solutions. The first step is to integrate Modelica models into Dynaplant to overcome its limits with respect to modeling flexibility (Sun et al., 2011). This development was started some years ago based on Dymola's code export feature. Then, FMI for Model Exchange was adopted as soon it became available. The use cases in chapter 2.1 explain this application in more detail.

Chapter 2.2 addresses the second class of use cases, which also deal with tool interoperability. However, in these cases co-simulation Functional Mock-up Units (FMU) are exported from Dynaplant models and serve as a representation of the real plant in the loop with a Nonlinear Model Predictive Controller (NMPC).

Chapter 3 focuses on the development of FMI support in Dynaplant and points out some issues experienced during implementation.

2 Use Cases

Two kinds of use cases are described in this chapter, one targeting the utilization of FMI for model exchange, the other addressing FMI for co-simulation, see Figure 1. Apart from different application areas major similarities exist:

- The use cases focus on tool interoperability
- All FMUs reside in-house only
- The models are huge with respect to number of parameters, dynamic states and internal/local variables

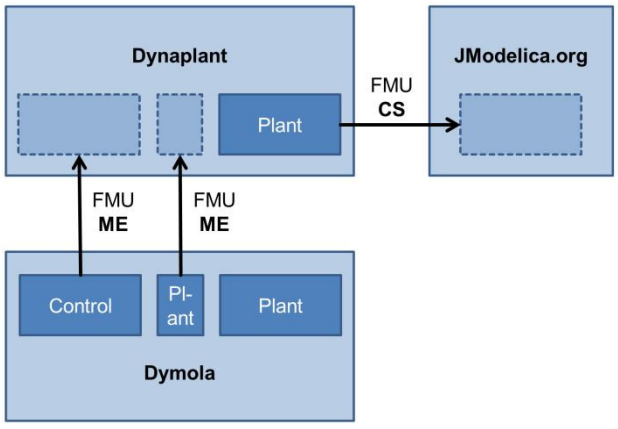

Figure 1: Overview of Use Cases

Our requirements of model exchange and tool exchange certainly differ in some aspects from those in

the use cases defined by the Modelisar project which are described in (Blochwitz et al., 2011). For our use cases there is no necessity for information hiding or IP protection. On the contrary, the goal is to keep as much information available for the user as possible.

2.1 Dynaplant Simulation

Our in-house simulator Dynaplant has been developed for more than ten years in a cooperation of Siemens Power and Gas and Siemens Corporate Technology. In the past, the analysis focused on the detailed, dynamic behavior of the water-steam cycle in a combined-cycle power plant.

A special emphasis has been drawn on the investigation of hydro-thermal dynamic-stability of once-through evaporators between gas and water side (Franke, 2008). Lately, the scope was extended to investigations on the plant level. In general, this leads to a higher need for flexibility in modeling because different subsystems come into focus depending on the application. FMI is an enabling technique to provide that flexibility, e.g. to add the gas side and also advanced controls, whilst keeping the efforts low. The simulation support of tight project schedules was only possible through the usage of FMUs.

In Dynaplant, components are modeled in one-dimensional resolution in an acausal way. The GUI is written in C# using a Microsoft Visio Add-in whereas the plant model itself is stored in a Modelica similar format. For simulation the plant is translated to C++.

2.1.1 Physical Component Models

The most natural use case for FMI in in-house simulators is to import models from different sources via model exchange. By this means, we integrate Modelica models of subsystems in Dynaplant to extent its application scope. The obvious driver behind this is the wish to benefit from both, the high performance of the C++ based in-house simulator and the modeling flexibility of Modelica.

Figure 2 shows the Modelica model of a multistage pump system controlled by a variable gear modelled in Dymola. The model has been prepared for FMI export which required an expansion of the physical fluid ports to a signal interface, already resolving the causality. Even for the very simple interface of such a pump system the transformation to a pure signal interface adds some overhead. Additionally, it becomes harder to understand the model when importing it as an FMU, see Figure 3. In Figure 4 the same system is modelled based on built-in components. The limitation of FMUs to signal interfaces very much hinder a convenient use as an imported component. These issues will boost the demand to provide all needed models as built-in components and not as FMUs. All the more, since the pump example is a rather simple system with a very small interface regarding the number of physical connectors. Looking at more involved power plant components like a drum model with multiple fluid connections across the boundary the graphical representation of the FMU holds almost no benefits because it becomes impossible to grasp the purpose of the model at first glance.

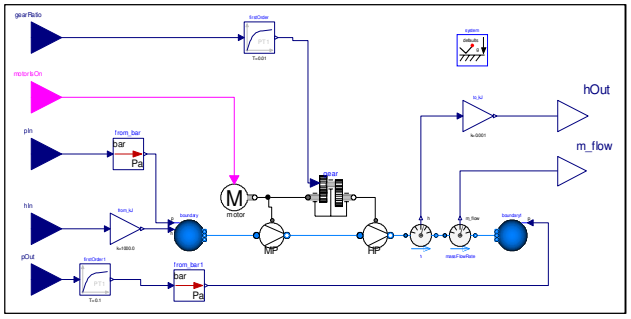

Figure 2: Modelica model of pump system

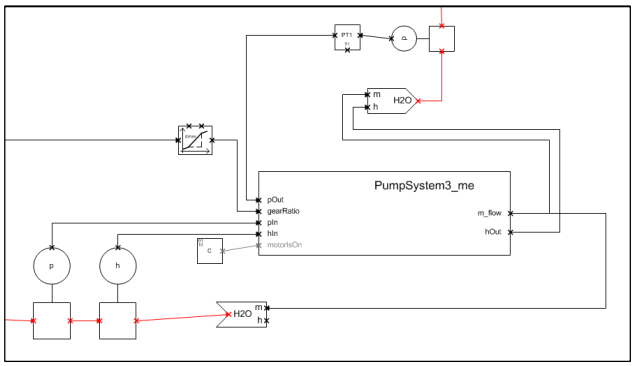

Figure 3: Integration of pump system FMU in Dynaplant

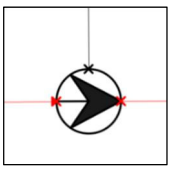

Figure 4: Dynaplant built-in model of a pump system

Besides the loss of information caused by the unstructured signal interface modeling, the lost structure of parameters in an FMU is the most severe shortcoming for us. Particularly, FMUs based on Modelica models make use of parameter hierarchies on many levels. Additionally, almost all professional Modelica libraries use features like grouping and tabs to generate a convenient user interface. Once exported to an FMU all this information is lost and the user is confronted with an unsorted list of parameters.

DOI 10.3384/ecp1511817

2.1.2 Controller in Dynaplant (based on generated Modelica code)

For testing plant control systems, Modelica models are automatically generated based on proprietary controller descriptions (Link et. al., 2014). In order to use these controller models in Dynaplant, Modelica models can be exported as FMU and imported into Dynaplant. Tests showed that large controller models are difficult to handle as FMUs for two reasons.

First, they use a bus connector (expandable connector) for signal exchange in Modelica. This bus connector contains hundreds of variables. Figure 5 sketches the concept for signal exchange. A global name space is replicated by using an expandable connector as an inner/outer component. Therefore, actuators and measurements can be placed in the plant on any level in the hierarchy. Furthermore, all boundary conditions are set on the bus connector. This graphical representation of the tested system as well as the underlying control layout is lost when using FMI.

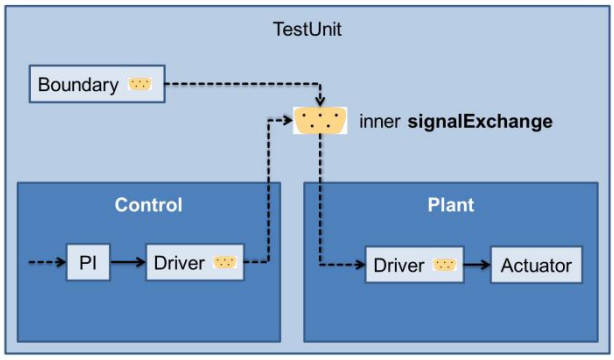

Figure 5: Signal Exchange in Modelica

All exchanged signals need to be available upon FMU import in Dynaplant in order to investigate control behavior. Currently, only scalar connectors are supported by FMI, which makes it hard to handle large sets of bus connector signals after importing the FMU. Future versions of the FMI specification might eliminate this problem.

Second, the controller models include many instances with a huge number of parameters, which are exported into the FMU. The resulting modelDescription.xml measures more than 150 MB. Compared to the Modelica implementation measuring about 6.7 MB (total model with all classes), this leads to slower model import and instantiation. As the compressed FMU appears to be small for the end user, potential causes for performance issues are not transparent. So far, we do not have a solution for providing a compact FMU interface with all required information. Limiting the number of visible parameters on the Modelica side cannot be intended, as they might be useful for investigating control behavior.

2.2 Offline Test of NMPC Loop

This section shows the usage of FMI for offline test of a Nonlinear Model Predictive Control (NMPC) loop. The basic concept of NMPC is to use a dynamic model to forecast system behavior and optimize the forecast in producing the best decision. In practice, an optimal control problem is solved over a finite future horizon, but only the first optimal control signal is applied to the system. Then the optimization horizon is shifted and the calculations are repeated. The solution of the optimal control problem depends on the initial state of the model which is the current state of the plant. In general, measurements are disturbed by noise or are missing, resulting in the need for a state estimation algorithm to determine the initial states under consideration of the past record of measurements.

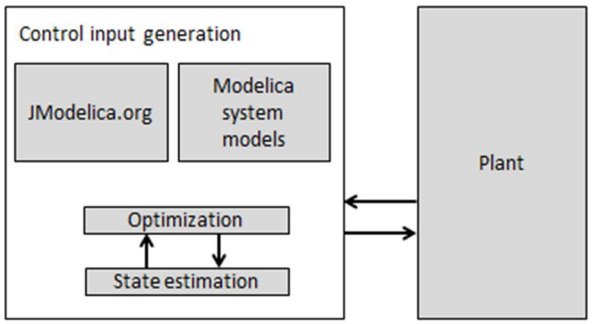

Figure 6: NMPC Loop

The model-based optimization framework looks as follows (see Figure 6):

- Modelica system models are used to describe the dynamics of the process and are used for optimization and state estimation
- The optimization is solved using JModelica.org, the open source platform for optimization, simulation and analysis of complex dynamic systems. JModelica.org interfaces the numerical solver IPOPT and CasADi – the framework for efficient evaluation of expressions and their derivatives
- For online application the optimal control signal is applied to the plant using the OPC interface of the Siemens control system SPPA-T3000 that performs all power plant automation tasks
- For offline tests pseudo measurement data is used to estimate the current state of the model. These measurements are generated by simulating a detailed model of the power plant including the control structure provided as co-simulation FMU exported from Dynaplant

The FMU has an internal state consisting of all values that are necessary to continue a simulation. This

feature could be used for NMPC to restart simulations after initialization with different values of the input signals. Unfortunately, the FMI 2.0 capability *canGetAndSetFMUState* is not yet implemented in JModelica.org. A workaround was implemented to set a consistent internal state, consisting of the values of continuous-time states as well as the iteration variables. It is worth noting that missing FMI capabilities or incomplete implementation is not the exception, but the rule. For each and every intended use case it is up to the user to test the capabilities of the chosen tool.

2.3 ControlTest in Modelica

This last use case is similar to the one from section 2.1.2, with the notable difference that not only control code is available in Modelica but also the plant model.

It is intended to show that the openness and flexibility of native Modelica models should not be underestimated compared to FMI. FMUs, as a translated and compiled version of the model, have well-known drawbacks e.g. when changing interface definitions or trying to understand the internal hierarchical model structure.

On a first look, coupling real-world controller code with physical plant models is a classic use case for FMI. Generally, there is a clear interface between the continuous plant model and the discrete controller implementation.

However, for testing power plant control systems, we currently prefer to have the full system in one uniform Modelica model. Although this requires transferring the graphical structure of control diagrams as well as the full implementation of all control blocks to Modelica. More details on this approach can be found in (Link et al., 2014).

For each study, specific plant models are built, specifically designed for the scope of the control task to be tested. Before starting the first simulations, a considerable amount of work goes into finding consistent boundary conditions for the relevant parts of the plant.

Having one homogeneous test unit in Modelica revealed the following benefits. The interface between plant and controller is not statically defined at the beginning. This allows fast adaptions to new controller strategies. Changes in controller or plant model do not require exporting new FMUs. Even though exporting FMUs requires little effort, ensuring to have the right FMU at the right place, creates unnecessary overhead during model development.

One holistic Modelica model allows control engineers and plant engineers to look into the same model. They both have access to the full system structure, equations and model-integrated documentation. This helps both sides to understand the system behavior and therefore to trust on the results. This reflects the special situation of usage of FMI in in-house applications, as already mentioned above.

Regarding documentation, most real-world FMUs currently lack on this part (e.g. documentation of model limitations and expected combinations of parameters and input signals). The FMI specification allows documentation to be added by the exporting tool. But, when exporting, it is not guaranteed that the documentation will be provided to user on the importing side. One needs to establish a process on how and where to store the binary FMUs together with the model source code. In contrast, using Modelica, we have one single source of truth, being the Modelica code in version control with hierarchical, model-integrated documentation.

The mentioned drawbacks of FMI are characteristically for any model exchange interface, therefore we do not propose to fix the FMI standard. Instead, we want to encourage users to carefully investigate a pure Modelica solution before introducing FMI interfaces in the tool-chain.

3 Implementation of FMI Support in Dynaplant

With respect to FMI, we support version 1.0 and 2.0 of model exchange import as well as 1.0 of co-simulation export. Implementation efforts are hard to estimate as they naturally depend e.g. on the experience of the implementer and also on the amount of optional features that shall be supported. However, we want to describe briefly in the following the main steps and issues of FMI implementation that we experienced in Dynaplant.

3.1 Import FMI for Model Exchange 1.0

Using an FMU in our tool requires that it can be handled almost as any other component. This means that it has an icon that can be dragged and dropped from a library onto the plant view. Furthermore, the component`s inputs / outputs do have graphical port representations that can be connected to ports of other components. Moreover, a parameter dialog needs to be available.

In order to support the described user experience, a gray box component has been added to the component library. After instantiation it shows a parameter dialog for specifying a FMU zip archive. The modelDescription.xml is parsed in C# where we generated a class from the available xsd scheme for deserialization. Parameters are then known if revealed by the FMU and it is possible to draw ports and the final component shape. FMUs in version 1.0 and 2.0 do not transport graphical information. Thus, own arrangements have to be done. In our software components usually have predefined ports and shapes and it required significant effort to build the final

DOI 10.3384/ecp1511817

component shape only during plant editing. However, after this specification step no extra effort is needed during plant editing for an FMU component, compared to any other component in the plant.

Stepping into simulation, we need to parse the xml-File again in C++, as much more static model information (e.g. information on internals and states) is required during run-time. It did not seem reasonable to overload already our model file (the plant definition) with all information and transfer it to C++. The implementation of calling the FMU functions with the correct arguments and in the right order was manageable with the help of the FMI specification and the FMU SDK of QTronic in mind. As FMUs of version 1.0 cannot transport information on the Jacobian, entries have to be calculated numerically which is quite extensive.

As far as our experience goes, a first implementation of model import can be set up rather easily, but testing and bug fixing is quite time consuming. It is often hindered by tool specific problems like license issues with the exporting tool (even if all partners have valid licenses) or unreasonable start values for inputs in the modelDescription.xml (e.g. some tools give only 0.0 for doubles). These values should be such that they allow for a successful and useful initialization if actually used and not overwritten during the import. Moreover, a general issue in testing is given by the nature of FMI as it is not possible to debug into the FMU to better understand what is happening inside a function call.

3.2 Import FMI for Model Exchange 2.0

Adopting FMI 2.0 has been accomplished by updating the implementation for FMI 1.0 which has been described in the section 3.1. The main effort consisted of updating of the XML parser and of our library of associated convenience functions in C++.

We now support an alternative way of calculating the Jacobian in case the capability flag *providesDirectionalDerivatives* is set to true. We need to provide also an alternative treatment in case an FMU does not support this capability. Generally, the FMI 2.0 specification and also implementation is complicated due to the large number of optional features, in particular capability flags. Two FMUs of version 2.0 and possibly from the same exporting tool can actually be very different in scope and capability. This becomes apparent only in the specific modelDescription.xml files.

With respect to performance, some firsts tests were done in comparing the import of the same Dymola model as FMU of version 1.0 and 2.0 (Dymola 2016 including bug fix on directional derivatives and sampling). Using the numerical calculation of the Jacobian entries, we experienced a significant decrease in performance between 1.0 and 2.0. For a rather small example with 92 algebraic and 1322 differential equations in total and 4500 s simulation time we measured a +11 % time usage in the DAE solver. Astonishingly, the loss of time was mainly in Jacobian calculation, although the FMU contributes no states but 310 events. Using the directional derivatives and the associated sparsity information, the performance decreased even further by roughly the same amount. For another example which actually contributed states, the performance of 1.0 and 2.0 roughly leveled up when using directional derivative information which proved to be a benefit in 2.0. Up to this point, a lot of questions remain and further tests are clearly indicated with various types and sizes of models and different exporting tools. It is not fully clear if import or export generate the issues.

3.3 Export FMI for Co-Simulation 1.0

Probably due to the different nature of weak and strong coupling in general, the effort to implement co-simulation export in our tools was significantly smaller than for model exchange import.

Most of the work was consumed in revealing plant information on all inputs / outputs, internals and parameters in a suiting way and offer access via *fmiGet…* and *fmiSet…* functions. The compliance checker was very helpful in bringing the modelDescription.xml in a correct way. It is quite remarkable that our model file of a Modelica similar format consumes roughly more than a factor of 8 less in storage than the modelDescription.xml including only inputs/outputs and parameters but no internals. We derived the learning that it makes sense to look into the FMI ticket trac for recognized issues in the specification, e.g., to find out that some importing tools expect the path to unzipped fmu folders in the call of *fmiInstantiateSlave* whereas others expect it to the zipped path, for instance.

We found testing and bug fixing very difficult and time-consuming also for co-simulation export. As Dynaplant is an in-house simulation tool we do not take part in official FMI cross-checks which could give a hint on still existing bugs but would probably not provide significant details on the root of the issues. It is often hard to track if issues occur in the FMU export implementation or in the importing environment. For the latter, the source code is usually unavailable and debugging possibilities or deeper knowledge on its communication handling are missing.

An important point in our internal discussions is the required resources of a co-simulation FMU. It is necessary to not only transfer one dll but around 35 dlls which are required by our simulator. To begin with, it is not clear where these should be copied to upon import such that the importing environment can

find them because all but one are loaded only implicitly. Any copy requires administrator rights at the target location and upon import, an FMU itself only knows target locations relative to its own unzipped location or the application path. If the latter is chosen, serious problems can occur if dlls with an identical naming can be found there already, potentially of different versions. Moreover, we face some difficulties in unloading all dlls from the address space after a run. The FMI description does not give guidance in dealing with such questions.

4 Summary

Even if the implementation of FMI support in in-house simulation tools implies a great effort, it is useful for many different applications as shown for some use cases in this paper. In principle, FMI can help or even enable some of the shown examples and some of our target applications, but still we face several limitations.

The use cases of chapter 2.1 highlight the urgent need to further develop the FMI standard with respect to the interfaces of FMUs. The existing scalar signal interface is definitely not powerful enough to allow the convenient application of FMI. Either it is to allow the implementation of "physical" connectors as needed in the use case described in chapter 2.1.1. Or to allow structuring of a huge number of signals as described in chapter 2.1.2.

A future FMI standard perfectly suited to our in-house applications would also need to support the concepts of acausal modeling - similar to the built-in behavior of Dynaplant and the basic principles of Modelica. Moreover, we face a mismatch between the intention of FMI to hide information and a need to reveal as much information as possible for in-house application.

Acknowledgements

The work for this paper was partially funded by the German Ministry BMBF (BMBF funding code 01IS12022A) within the ITEA2 project MODRIO.

References

Franke, J., Brückner, J. (2008): Dealing with tube cracking at Herdecke and Hamm-Uentrop, *Modern Power Systems*, October 2008.

Christian Bertsch, Elmar Ahle, Ulrich Schulmeister (2014): The Functional Mockup Interface - seen from an industrial perspective, *Proceedings of the 10th International Modelica Conference*, March 10-12, 2014, Lund, Sweden. doi:10.3384/ecp1409627

Kilian Link, Leo Gall, Julien Bonifay, Matthias Buggert (2014): Testing Power Plant Control Systems in Modelica, *Proceedings of the 10th International Modelica Conference*, March 10-12, 2014, Lund, Sweden. doi:10.3384/ecp140961067

Yongqi Sun, Stephanie Vogel, Haiko Steuer (2011): Combining Advantages of Specialized Simulation Tools and Modelica Models using Functional Mock-up Interface (FMI), *Proceedings of the 8th International Modelica Conference*, March 20th-22nd, Technical University, Dresden, Germany. doi:10.3384/ecp11063491

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olßon, J.-V. Peetz, S. Wolf, C. Clauß (2011): The Functional Mockup Interface for Tool independent Exchange of Simulation Models, *Proceedings of the 8th International Modelica Conference*, March 20th-22nd, Technical University, Dresden, Germany. doi:10.3384/ecp11063105