



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Integration of Functional Mock-up units into a Dynamic Power Systems Simulation Tool.

P. Top, L. Min, Y. Qin

November 10, 2015

IEEE PES General Meeting 2016
Boston, MA, United States
July 17, 2016 through July 21, 2016

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Integration of Functional Mock-up units into a Dynamic Power Systems Simulation Tool

Philip Top, Yining Qin, and Liang Min
Lawrence Livermore National Lab
Livermore, California 94550
Email: {top1,qin3,min2}@llnl.gov

Abstract—Modelica is an object oriented modeling language that allows straightforward specification of differential, algebraic and discrete equations in a standardized format. It has been used in a number of applications to support multi-domain system simulations such as are found in the automotive and aerospace industries. More recently it has begun to be used in the power systems industry. Modelica models can be compiled into functional mock-up interfaces(FMI) which contain standardized interfaces to the modeling equations and dynamically linked libraries which allow the coupling of models with each other or into other simulation software. The interface allows the decoupling of the model from the mathematical solver used for the simulation. In this document we describe in detail the coupling between functional mock-up units and Griddyn, a dynamic power system simulation tool, which incorporates a dae solver and variable time step methods.

Index Terms—Simulation, High performance computing, Modelica, DAE, functional Mock-up interface, dynamic simulation

I. INTRODUCTION

Simulating power systems requires models of transmission networks, loads and generators. These models produce a set of nonlinear differential algebraic equations(DAE) that must be solved to arrive at the solution for a particular time instance and integrated through time to produce a dynamic time series. The models for generators and loads can range from simple to very complex, and they must be specified in detail to produce an accurate simulation. Commercial software such as PSS/e and PSLF contain many such models for generators and control systems and other software does as well, and they may optionally include a some language to describe additional models. However these models are still part of the proprietary code base of these packages, and, though the model descriptions are available in block diagram format, differences still exist between the packages in the implementation details. These differences make comparison difficult and tedious and prevent the utilization of these models in other contexts, or the transfer of models between software systems. Several proposals and projects are underway to better standardize power systems models and libraries through the use of the open Modelica language including efforts to integrate with the common information model(CIM)[1] and build a library of power systems models [2] a the power systems library [3]. Through the effort described here we hope to provide

an example of how FMU's from these libraries and elsewhere can be integrated into a power systems package.

A. Modelica

Modelica is a free object-oriented mathematical modeling language for describing large-scale and multi-domain physical systems. Through its hierarchical language it can support complex systems from across a wide spectrum of physical domains such as electric, thermal, mechanical, communication, hydraulic, and other control systems The Modelica's key features include hierarchical components modeling with actual physical structure without data flow direction definition, code re-usability, and decoupling of models from solvers. Modelica is also supported with a number of standardized and highly specialized libraries in a variety physical domains.

Several commercial packages are available for Modelica development including Dymola(Swedish company Dynasim AB), AMESim (LMS International), CyModelica (American company CyDesign Labs), Wolfram SystemModeler (Swedish company Wolfram MathCore AB) and others. Dymola is the most popular commercial Modelica software which consists of libraries and components from many engineering domains for mechanical, electrical, control, thermal, pneumatic, hydraulic, power train, thermodynamics, vehicle dynamics, air conditioning, etc[4]. There are also two main open source Modelica platforms: OpenModelica and JModelica.org. OpenModelica is Maintained by the Open Source Modelica Consortium (OSMC- a non-profit organization) [5] is a set of integrated platform including virtual modeling editor, result monitor and solvers for industrial and academic usage. JModelica.org is an source platform for simulation optimization of Modelica models, while offering a flexible platform serving as a virtual lab for algorithm development and research [6].

The development of complex energy systems such as the smart grid requires advanced modelling tools. Modelica is one such tool that can help meet the requirements of current and future system developers[7]. Modelica has been successfully applied to complex energy system simulation such as power generation, building energy efficiency, power grid protection, and smart grids [8], [9], [10]. Modelica provides mechanical and electrical components adapted to smart grid simulation, and a multi-agent approach for supporting co-simulation plat-

forms to connect several simulators by FMU (Functional Mock-up unit) [11], [12].

B. Functional Mockup Interface

Modelica code can also be compiled into C code or into a module that can be linked with other tools. This is accomplished through the functional mock-up interface (FMI). The FMI interface was formalized in 2011 with a major revision in late 2014[13]. The standard supports both model exchange and co-simulation. The models are packaged in a functional mockup unit (FMU) which consists of an xml model description file, a binary dynamic linked library, and any supporting files compressed into a single file. The original C or Modelica code can be included, but are not necessary. In this way, organizations can share proprietary models with other organizations and link them together in larger systems. Several of the tools that can be used for developing with Modelica can generate FMUs from Modelica code including OpenModelica and Jmodelica. FMI support is included in a number of different environments often used for modeling including Simulink, Labview, Python, Excel, MapleSim, and Ansys, in addition to many of the previously mentioned Modelica tools[?].

The FMI interface includes functionality for getting and setting system parameters, and the information necessary for evaluation of the system as part of the simulation. The XML file contains the model information that is not necessary for runtime execution such as variable names and dependency information.

C. GridDyn, a Power Transmission System Simulator

GridDyn, a transmission system simulator, is built using object oriented C++ methods to ensure modularity. It supports the functional separability of models, input-output, and solver technology to facilitate implementation and integration of new models, new algorithms, coupling with other simulation software, and execution on both large-scale HPC systems and desktop platforms.

Models in GridDyn are classified into three types: *primary objects*, *secondary objects* and *submodels*. Primary objects include areas, busses, links (such as transmission lines) and relays. For example, a relay in GridDyn is a conceptual object that can monitor, control and manipulate one or more objects in the system. Primary objects can contain other primary objects in a hierarchical structure. Secondary objects are those that attach to a bus, such as loads and generators. Submodels control the dynamic behavior of the other components and can be incorporated through any of the 3 classes. Mechanisms are also in place to instrument the system to extract desired information in log files or other communication paths, as well as manipulate the system through events and changes in the models (e.g. triggering a fault).

Numerous solution modes are available, including *DC power flow*, *AC power flow*, *stepped power flow*, and a *full dynamic solution*. The stepped power flow evolves the system through time by repeated evaluations of a power flow solution.

The dynamic solution includes limiting features in many of the models such as excitation limits, governor deadbands, and output limits. The software includes the capability of running under a cosimulation framework with Gridlab-d an open source distribution simulation platform developed by PNNL, and NS-3 and open source communication simulator[?].

As its default solver GridDyn uses the Implicit Differential Algebraic equation solver, IDA, for time integration of the transient system and the KINSOL nonlinear algebraic system solver for solution of power flow systems [14], [15]. Both of these packages are part of the SUNDIALS suite of codes and support both distributed and shared memory parallel execution [16], [17]. The IDA package uses a variable step and variable order method for implicit time integration. This method adapts the integration order and step size to ensure that the solution will satisfy specified accuracy tolerances. Hence, the solution returned is highly accurate and not subject to pollution from error generated by a large, fixed time step. The code exploits periods of low dynamics to take larger step sizes. In addition, IDA provides the capability to check for roots of specified equations at each time step. When these equations change signs within a step, IDA will find and return the exact time of the zero-crossing. Integration in IDA is conducted either to a specified stop time or to an earlier time when a root is found.

II. FMI INTEGRATION

For the most part, the operational sequence of the FMI interface matches closely with that used internally inside Griddyn. This similarity allows a straightforward mapping from the internal GridDyn interface to the function calls and sequences called for from the FMU. The FMI exposes the internal states corresponding to the ordinary differential equation portion of the solution, while keeping the algebraic components internal to the FMU. In FMI version 2.0, directional derivatives are also optionally exposed which will allow direct inclusion into the Jacobian calculations in GridDyn. For FMI version 1.0, and when not available for version 2.0 these the jacobian calculations are numerically approximated. The actual integration is done through a toolbox built by Modelon[?] the same company that maintains an open source Modelica environment JModelica. The toolbox is the FMI library 2.0 and forms the basis for an FMU checker tool. The library includes functions to extract the FMU, read and interpret the XML file and call the appropriate FMI functions, and handle the loading of the shared libraries used in the FMU. The library includes a Cmake build system, so it can be incorporated in the existing build system of Griddyn. While the interface is standardized, the actual details of the implementation can vary between different FMU's particularly in the matter of input and output causality, and the initialization details. These differences result in increasing complexity in the actual interface. The challenges associated with these differences has been noted in other efforts[?].

The actual implementation interface was captured in a Griddyn subModel. This submodel was then contained in other models in the system that contained additional interfaces and

details to match specific requirement of other Griddyn models. Example interfaces include loads, generators, governors, exciters and other grid specific controls. These higher level models matched up FMU inputs and outputs with the appropriate signals and states inside Griddyn so the model would interface properly. For instance, a load model in Griddyn takes three signals from the connected bus– the bus voltage, angle, frequency. Some or all of these signals can be used in the model to compute the real and reactive power consumed by the load. The FMUload model must direct these signals into the appropriate inputs of the FMU and allow for the system to extract the required partial derivatives.

A. initialization

Modelica code and hence FMUs can have a wide range of possible modes for initialization. In Griddyn the most common mode of operation is to start the dynamic simulation from a flat start with all $\frac{dx_i}{dt} = 0$. Some FMU's have functionality to accomplish this internally, others start from a 0 of fixed state and others have component specific behavior. In order to accommodate the varying methods of initialization Griddyn includes a few user controlled flags to manage the initialization. These give the option to use the internal initialization in the FMU, to specify an exact state, or to use the algebraic solver in Griddyn to force the FMU state to be a flat start.

As a general strategy Griddyn defines the system equations in terms of a residual function and solver tries to find an x and x' such that

$$f(x, x') = 0 \quad (1)$$

An equation of this form is defined for every system variable. For power flow problems x' is assumed to be fixed usually to 0. The problem is formulated as a nonlinear algebraic system and since the solver and models are logically separated in the code and the models independent at a given layer, it is possible to accommodate additional algebraic equations from the models that are not part of traditional power flow equations. In the case of using the algebraic solver for initialization, the fmi submodel defines an equation for each state of the system

$$f_i(x_i) = \frac{dx_i}{dt} \quad (2)$$

where x_i are the states in the fmu and $\frac{dx_i}{dt}$ are the derivatives as computed by the FMU. The solver iterates until a solution forcing the derivatives to 0 within the problem tolerances are found.

B. Jacobian Computation

As part of the solution for the algebraic solver and for the DAE system, Griddyn requires the computation of a Jacobian matrix. This requires calculation or at least an approximation of all $\frac{\partial f_i}{\partial x_j}, \frac{\partial f_i}{\partial z_j}, \frac{\partial y_i}{\partial x_j}, \frac{\partial y_i}{\partial z_j}$ for the fmu where x_i are the states, f_i is the residual function for the state x_i , z_i are the inputs and y_i are the outputs. The partial derivatives of the outputs may be required for computation of Jacobian elements elsewhere in the system. Griddyn includes 3 functions in each model for computing the various partial derivatives, one for $\frac{\partial f_i}{\partial x_j}$ and $\frac{\partial f_i}{\partial z_j}$,

one for $\frac{\partial y_i}{\partial x_j}$ and a third for $\frac{\partial y_i}{\partial z_j}$. The latter two are optionally called if the outputs are used in the residual functions for other states. In many systems the outputs are actually states which simplifies the computation but in others they are functions of the states and inputs. The FMI 2.0 standard includes an optional function for computing partial derivatives, however, since it is optional, it is not included in all FMU's, thus requiring functions for numerical computation of the Jacobian terms for the FMU.

The XML description file includes identification of all the dependencies in the model which is used in the Griddyn numerical computation of the jacobian to reduce the number of computations and function calls. The basic idea is to approximate the partial derivative with

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x_j + \delta, x, x') - f_i(x_j, x, x')}{\delta} \quad (3)$$

In the simplest case this is accomplished through a sequence of get and set calls using the FMU interface. However, this sequence is where deviations and differences among FMU's and the causality of the sytem become problematic. In some FMUs, the get functions do not trigger new computations and instead return the previously computed value even though it is technically no longer valid due to updated input values. The values may be flagged to be updated but the get does not trigger the new update. It simply returns whatever value is in the internal buffers. This situation requires that a call to get the derivatives or some other method of triggering the updates is required when new parameters are entered and again when the old parameter is restored. Further difficulties arise when getting the output from the FMU for the output partial derivatives calculations if the output requires an algebraic computation. In some FMU's the output is only updated at the conclusion of a time step, which poses significant difficulties for the Jacobian computation of the output dependencies on the inputs and other states. It should be noted that FMUs in this case are quite reasonable for for integration into an ODE system since the time steps are small and all components consistent at every time step and the derivatives consistent at intermediate time steps. Unfortunately, in the case of integration into DAE system using an iterative solver the intermediate outputs are part of a larger system and thus can be inconsistent if the FMU is not not set up to behave consistently with that scenario. It is hoped that future versions of the FMU specification might resolve the inconsistencies and allow specification of the required connections. To account for these variabilities in behavior in FMUs, Griddyn includes a number of probing functions and approximations to correct the potential inconsistencies.

Assuming a numerical Jacobian computation is required the system probes the fmu searching for the simplest set of calls to compute the Jacobian. This information is stored along with the dependency information. For system outputs if it is determined that the outputs are only valid immediately after a completed timestep an approximation mechanism is enabled. An alert is issued to the solver to notify the fmu controller after

every completed timestep of the solver and the output values are updated. For outputs required in intermediate timesteps an approximation is made:

$$y_i(t_0 + \delta) \approx y_i(t_0) + \sum_{j=0}^N \frac{\partial y_i}{\partial x_j} (x_j(t_0 + \delta) - x_j(t_0)) + \sum_{j=0}^M \frac{\partial y_i}{\partial z_j} (z_j(t_0 + \delta) - z_j(t_0)) + \frac{\partial y_i}{\partial t} \delta \quad (4)$$

where N is the number of states and M is the number of inputs. The $\frac{\partial y_i}{\partial t}$ term is only included if the output has an explicit dependence on the independent variable. The partials are numerically computed by repeated calls to the completedTimeStep function with different states and inputs. These values are updated periodically or if the values change sufficiently to warrant a new call. The error between the predicted and the newly computed value is usually used as a mechanism for determining whether or not to update the values. During a power flow solution, the direct approximation of the jacobian is used by repeated calls to the completedTimeStep function. This is done since time is remaining constant during that evaluation process and any nonlinear effects are more prevalent given the large potential range of the values during initialization. During dynamic time steps the linear approximation of the output between completed time steps shown in Equation ?? will result in a small interfacing error but will be controlled as the solver uses adaptive time steps depending on the rate of change of the system. From a system perspective, simply holding the output fixed between dynamic time steps will likely result in an acceptable solution in many situations. The linear output approximation reduces the potential error even further, in situations where the exact value is not available. User controls are available to force which mode to use if desired.

C. Dynamic Simulation and Event Indicators

As stated previously, Griddyn currently uses IDA as a nonlinear DAE solver, though it can accommodate other solvers if required. IDA includes a root finding function which evaluates a user defined function searching for sign changes in the function. If it detects a sign change, IDA begins a search routine to locate the exact time of the sign change and returns to the user to deal with the root. A number of the models in Griddyn make use of root finding to detect limit violations or other triggers that cause a change in the internal equations of the model. FMUs also implement this functionality in an event indicator. Detecting events in an FMU requires evaluation of the getEventIndicator function. Given the similarities between events in FMUs and roots in Griddyn, the mapping was very straightforward, and the interface in Griddyn becomes a wrapper to the FMI calls.

For computations concerning a state variable a residual function was created

$$f_i = \frac{dx_i}{dt} - x'_i \quad (5)$$

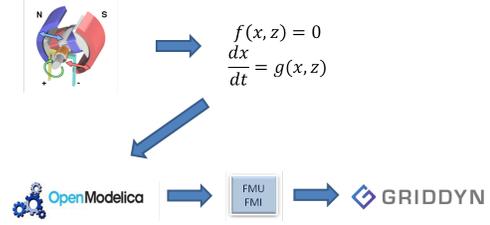


Fig. 1. Example workflow of modeling

where $\frac{dx_i}{dt}$ is from the FMU derivative evaluation and x'_i is the time derivative of the state as calculated by the solver. When the two estimates of the time derivative match the residual function goes to 0. The Jacobian is adjusted to include the x'_i term.

If an output approximation is necessary the solver is operated in single step mode and the states for the FMU finalized after each step to get a updated valid output. Otherwise the solver is allowed to continue until the specified end time, a root is found or a predetermined event is reached. At which point, the states of all system models are updated. If the event causes a change in the simulation the solver is reinitialized and computation continues. If the event is in the FMU itself the fmsubmodel switches the FMU to event Mode, the event is executed and fields and values are updated, new states estimated, and the fmu is switched back into continuous time mode for further computation.

III. EXAMPLE AND RESULTS

For testing purposes a number of FMUs were tested to ensure consistent and correct evaluation inside Griddyn. An example workflow is shown in Figure 1. A physical object is described by mathematical expressions, these expressions are coded in Modelica, in this case through the OpenModelica interface. OpenModelica is used to create and FMU and that FMU is utilized by Griddyn. As an example we modeled a simplified single cage motor model as a load. This model was implemented and tested in 3 contexts purely in Griddyn, purely in Modelica, and then run through an FMU in Griddyn. In one experiment the load on the motor was shifted at 1 second to a torque of 0.9 pu and back to 0.7 pu at 6 seconds. The result is shown in Figure 2. Shown are the Modelica and Griddyn results from the simulation in OpenModelica and through the FMU in Griddyn. The results were very close and the differences are shown in Figure 3. These differences are due to the differences in sample times and error tolerances in the different solvers and the variable time stepping in IDA. The reactive load showed a similar pattern. The results from the model implemented purely inside Griddyn were nearly indistinguishable from the FMU version.

Other loads models with varying degrees of complexity and features along with a simple governor model were tested with all results being similarly matching between their use inside Griddyn and through a Modelica software package. Thus demonstrating successful inclusion of FMU into a DAE solver system.

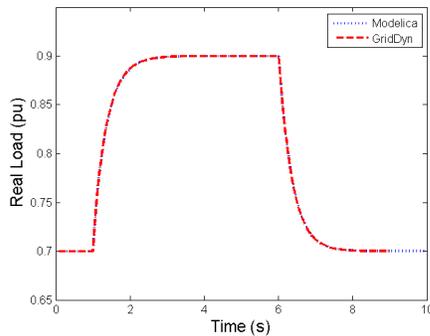


Fig. 2. Simulation results for Griddyn FMU vs Modelica

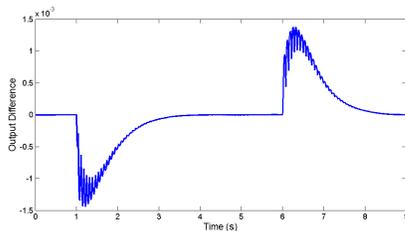


Fig. 3. Differences between Modelica results and Griddyn Results

Early testing indicates that operating through a FMU interface is somewhat slower than the native interface which is to be expected given the additional copies and more complex jacobian calculations. Further tests involving a large number of FMUs are planned to determine a more precise metric for the impact on simulation speed within a more controlled compilation environment and system.

IV. CONCLUSION AND FUTURE WORK

This article describes the detailed inclusion of Functional Mockup Units into a dynamic power system simulation engine based on a variable time step DAE solver. The differences between individual FMUs in the timing of updates and the causality of input and output variables leads to some approximations and additional management particularly around output variables in order to properly interface FMUs into the simulation tool.

Once enabled, the coupling of FMUs into a power system simulation tool allows for testing of complex systems as part of a larger network. The system models can be used in any number of tools and model consistency maintained between the various tools, it also allows the possibility of multi-domain models integrated easily into a power system simulation. The same model can be used as part of a detailed single machine model in a tool like Simulink and then replicated hundreds of times as part of a larger system study without worrying about differences in model interpretation between the various tools. The work done here shows how these models can be incorporated into other power system tools as well.

Future work will involve a more detailed and rigorous study of the performance impacts of using FMUs on the simulation

in isolation and with multiple copies. Some thought is also being put into wrapping Griddyn itself as an FMU for co-simulation to enable more options for coordination with other tools through and FMI interface and make power system modeling as easy, flexible and reliable as possible.

ACKNOWLEDGMENT

This work was supported by the Department of Energy's Advanced Grid Modeling (AGM) program. The work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

- [1] F. Gomez, L. Vanfretti, and S. Olsen, "Binding cim and modelica for consistent power system dynamic model exchange and simulation," in *Power Energy Society General Meeting, 2015 IEEE*, July 2015, pp. 1–5.
- [2] T. Bogodorova, M. Sabate, G. Leon, L. Vanfretti, M. Halat, J. Heyberger, and P. Panciatici, "A modelica power system library for phasor time-domain simulation," in *Innovative Smart Grid Technologies Europe (ISGT EUROPE), 2013 4th IEEE/PES*, Oct 2013, pp. 1–5.
- [3] R. Franke and H. Wiesmann, "Flexible modeling of electrical power systems—the modelica powersystems library," in *Proceedings of the 10th International Modelica Conference*, March 2014.
- [4] "Dymola," <http://www.modelon.com/products/dymola>.
- [5] "Openmodelica," <https://www.openmodelica.org/>.
- [6] "Jmodelica.org," <http://www.jmodelica.org/>.
- [7] A. Elsheikh, E. Widl, and P. Palensky, "Simulating complex energy systems with modelica: A primary evaluation," in *Digital Ecosystems Technologies (DEST), 2012 6th IEEE International Conference on*, June 2012, pp. 1–6.
- [8] M. Richter and F. Mollenburck, "Flexibilization of coal-fired power plants by dynamic simulation," in *Proceedings of the 11th International Modelica Conference*, September 2015.
- [9] P. Eberhart and T. S. Chung, "open source library for the simulation of wind power plants," in *Proceedings of the 11th International Modelica Conference*, September 2015.
- [10] M. Bonvini and M. Wetter, "A modelica package for building to electrical grid integration," in *Fifth German-Austrian IBPSA Conference RWTH Aachen University*, September 2015.
- [11] O. Chilard and J. Boes, "The modelica language and the fmi standard for modeling and simulation of smart grids," in *Proceedings of the 11th International Modelica Conference*, September 2015.
- [12] J. Enerbck and O. N. Nilsson, Master's thesis.
- [13] M. consortium and M. A. P. FMIi, "Function Mock-up Interface for Model Exchange and Cosimulation," https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf, specification, July 2014.
- [14] A. C. Hindmarsh, R. Serban, and A. M. Collier, "User documentation for IDA v2.8.0," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-SM-208112, 2015.
- [15] A. M. Collier, A. C. Hindmarsh, R. Serban, and C. S. Woodward, "User documentation for KINSOL v2.8.0," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-SM-208116, 2015.
- [16] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 363–396, 2005.
- [17] "SUNDIALS (SUite of Nonlinear and Differential/ALgebraic Solvers)," <http://www.llnl.gov/casc/sundials>.