MODELICA-ENABLED RAPID PROTOTYPING VIA TRNSYS

Atiyah Elsheikh, Edmund Widl, Peter Pensky, Florian Dubisch,
Markus Brychta, Daniele Basciotti and Wolfgang Müller
Austrian Institute of Technology, Vienna, Austria

Emails: surname.familyname@ait.ac.at

ABSTRACT

Traditional building simulation tools have achieved considerable success in the past. They provide the essential foundation for modeling highly sophisticated tasks. Nevertheless, new challenges and current progress in the energy domain require rapid prototyping capabilities for just-in-time model-based investigation. Supporting these requirements is one of the many advantages of employing modern universal modeling languages. This work addresses the integration of the modern modeling language Modelica with the traditional simulation tool TRNSYS. Using the modern standard functional mock-up interface for tools interoperability, a straightforward way for Modelica-enabled rapid prototyping within TRNSYS is presented.

INTRODUCTION

Traditional simulation tools in the new era

Many existing traditional simulation tools achieved a profound state for highly-reliable sophisticated modeling applications in the buildings domain. They are based on decades of conceptualization and progressive developments. The practicality of such tools is indicated by the corresponding large user communities and the cooperation efforts among different working groups. They typically provide a large set of intensively tested model components out of which systems can be assembled and simulated. Nevertheless, coping with future-oriented concepts, new research-oriented ideas and the ever more emerging technologies still represent a challenging aspect and a realistic obstacle for such traditional tools. Modelers are rather dependent on the available set of components provided by their favourite tools.

For instance, TRNSYS[1] (Klein et al., 1976), a specialized simulation tool for modeling the thermal behavior of buildings, can be named as an example. While it provides low-level functionalities for developing additional components, the implementation of further desired complex components using classical programming languages like Fortran and C++ becomes a complicated task in terms of efforts for most programmers.

The rapid development within the Energy domain in general emphasizes the importance of providing rapid prototyping capabilities for modeling emerging technologies before they get built. In particular, building simulation applications increasingly require the interactions of many components from multi physical domains (e.g. renewable energy resources, intelligent control strategies and communication with other units within smart grids, etc.). The organization of such components within hierarchies of subsystems necessitates flexible descriptive capabilities and high-level programming paradigms, e.g. hybrid systems. These are the features that are best supported by advanced universal modeling languages.

Modern modeling languages

An increasingly followed approach is to employ modern modeling languages such as Modelica (Elmqvist and Mattsson, 1997). Modelica relies on powerful modeling concepts with which complex systems can be rapidly prototyped. Object-oriented facilities and powerful descriptive syntax allow for model components reuse, hierarchical system decomposition and object inheritance (Elsheikh et al., 2012). Existing standardized libraries, e.g. in thermodynamics, fluid dynamics and others, provide the base for modelling highly-specialized applications, e.g. the Building library (Wetter, 2009). According to the experiences reported in (Wetter and Haugstetter, 2006), prototyping applications with Modelica is five to ten times faster than with TRNSYS.

Nevertheless, while the adopted universal modeling concepts are ideal for modeling multidisciplinary applications, the absence of domain-specific concepts leads to some limitations of the applications scope in comparison with traditional specialized simulation tools. Namely, domain-specific concepts allow for high-level implementation of detailed components that can not be easily reproduced with Modelica.

Contribution

In this work, we combine the advantages of both types of tools,TRNSYS and Modelica. By enabling the integration of Modelica-based types within TRNSYS, the advantages of employing the highly specialized modeling capabilities of TRNSYS are combined with the ability of performing rapid prototyping within Mod-

[1] `http://sel.me.wisc.edu/trnsys/`

elica. In this way, the scope of building simulation applications can be extended. In this context, we make use of the Functional Mock-up Interface (FMI)[2] (Blochwitz et al., 2011), a modern unified model interface for model exchange and tool interoperability. We show the details and the potentials of extending the capabilities of TRNSYS with the Modelica language via FMI. Particularly, the merging of the different underlying modeling approaches is addressed.

RAPID PROTOTYPING VIA MODELICA

The acausal modeling approach

The complexity of modeling a large-scale system is practically reduced by decomposing it into, more or less, a finite set of basic components. Each of these components, characterized by a relatively small equation system, is implemented, tested and maintained in a faster way. A system is assembled by connecting these components together. For that purpose, well-defined connection mechanisms have to be realized for establishing a meaningful interpretation of connected components. One of the classical approaches is the block-diagram approach where some output variables of a component block become the input variables of another component block. For instance, this approach is followed by Simulink[3] and TRNSYS.

An alternative way is the acausal modeling approach, where the causality among model components is usually absent. The key issue behind that approach relies on fundamental conservation laws of Physics, e.g. conservation of energy. Namely, the sum of all flows of energy (or mass, momentum, etc.) at a certain point in a closed system sums to zero (Fritzson, 2003, Sec. 14.1, P. 477), see Figure 1. Based on these fundamental principles, each model component provides interfaces called connectors, which work as communication ports to other connected components. Such interfaces are usually characterized by two types of variables:

1. Flow variables E as energy carriers, e.g. heat transfer rate, current, flows, etc.
2. Potential variables P measuring energy levels, e.g. temperature, voltage, pressure, etc.

The choices of connector variables depend on the physical domain, e.g. temperature and heat transfer for thermodynamics and voltage and current for the electrical domain, respectively. A connection point between connectors represents two types of equations:

1. A sum to zero equation for flow variables
2. An equality equation for potential variables

These equations define how energy (or mass, momentum, etc.) propagates among connected components.

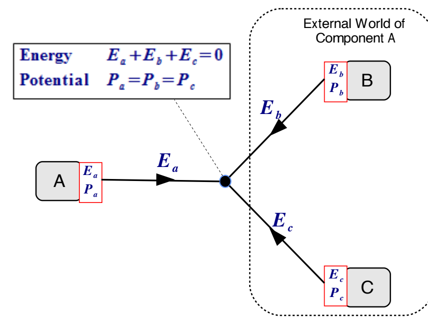

Figure 1: Connection mechanism of acausal modeling

Thus, a system is not viewed any more as a set of interacting components with explicit input/output relationship. Models are assembled in a way that look very similar to the conceptual reality. Modification, insertion and deletion of model components become much easier to realize as they do not lead to significant changes in the structure of the assembled model.

Modelica background, features and current state

The acausal modeling approach pioneered by (Elmqvist, 1978) had a revolutionary impact on the modeling community. Many simulation tools had been implemented upon, e.g. VHDL-AMS (Ashenden et al., 2003) and gPROMS[4]. Nevertheless, the variety of existing tools had negative aspects. While each of these tools had its own features and strength, it was not possible to exchange models among different tools. Moreover, a lot of conventional efforts had been multiply realized by each tool (Åström et al., 1998). To overcome these drawbacks, the development of the Modelica language specification was initiated for unifying this splintered landscape of modeling languages (Mattsson and Elmqvist, 1997). Through intensive discussions among many involved participants from academia and industry, the following main features were adopted within Modelica:

- an open-source model-exchange specification that can describe small pieces of complex systems and their interrelationship
- a causal modeling approach as well as other relevant modeling paradigms and promising features (e.g. object-oriented facilities) provided by existing modeling languages
- domain-neutral concepts adequate for multidisciplinary applications
- an equation-based syntax

A distinguished feature of Modelica is the employment of equation-based syntax. This adds another dimension of non-causality. In contrast to typical assignments which express a clear relation between an output and a set of inputs, equations express relations among variables that need to be fulfilled concurrently. Equations can be written in an implicit way and there is no

[2] www.functional-mockup-interface.org
[3] www.mathworks.com
[4] http://www.psenterprise.com/gproms/index.html

need to place them in a specific order. Another significant potential advantage of Modelica is the presence of a large set of free and commercial libraries in many physical domains within an ever growing Modelica Standard Library (MSL) maintained by the Modelica Association (MA)[5]. These libraries can be the basis for highly sophisticated applications. The MA is also responsible for further development and maintenance of the open-source specification of Modelica. A periodical international conference on Modelica is organized with ever growing participation from academia and industry.

Compiling and simulating Modelica models

While many models can be easily described using Modelica high-level syntax, it is the responsibility of common Modelica compilers, e.g. OpenModelica (Fritzson et al., 2006), Dymola (Brück et al., 2002) and JModelica (Åkesson et al., 2010), to translate such models into simulation code. The acausal modeling approach results in typically large-scale equation systems even for relatively small models. Consequently, motivated by the evolve of modeling languages, many tools and algorithms based on graph theory have been developed for representing, manipulating and simplifying such systems (Cellier, 1991; Maffezzoni et al., 1996). Typical tasks that are performed by a Modelica compiler include but are not limited to:

1. index reduction of Differential Algebraic Equations (DAEs) (Pantelides, 1988)
2. providing reliable algorithms capable of computing consistent initial conditions of state variables (Bachmann et al., 2006)
3. computing accurate sparse Jacobians using algorithmic differentiation techniques for performing stable numerical integration by state of the art numerical solvers (Braun et al., 2011)

All these efforts allow the modeler to focus more on the modelling task without paying attention at low-level mathematical details.

Example

Figure 2 demonstrates a network of pipes model taken from the examples subpackage within the `Modelica.Fluid` standard library (Franke et al., 2009). A medium is supposed to flow from the source to the sink regulated by the valves. The diameter and the length of each pipe is parametrized in the model. Branching and Junction of fluid flows can be efficiently handled by Modelica. The concepts behind the connection points guarantee the conservation of energy, mass and momentum of the fluids flow. Issues like reverse flow and connection of pipes with different diameters can be efficiently treated. The model facilitates the capabilities of Modelica for performing one-to-one mapping of real large-scale systems into a set of connected components. The model is assembled by just copying, dragging and connecting icons together. It is principally straightforward to modify the architecture of such multi-way connections for achieving optimal design. More insights into some elements and language constructs of the Modelica language can be consulted in (Elsheikh et al., 2012)

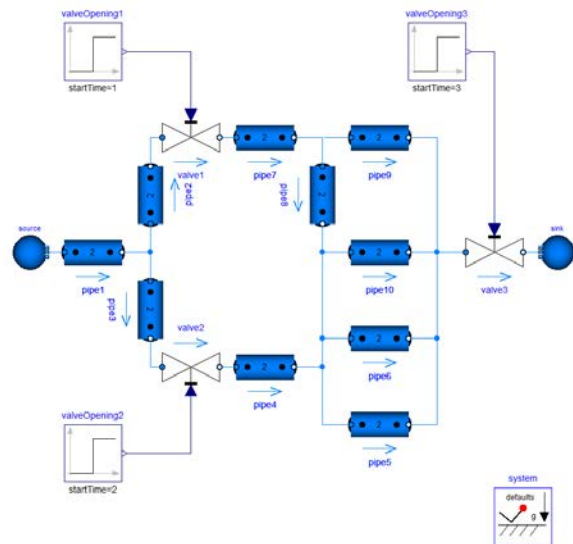

Figure 2: A multiway connections of pipes from a source (the circle in the left) to a sink. The flow of a medium is regulated by a set of valves. The valves are opened and closed according to explicitly given times.

THE FMI

Background

FMI is a standardized unified model interface for co-simulation and data exchange of model components between simulation programs. It is a result of the MODELISAR project[6] and it is further maintained and improved by the MA. A variety of software already supports FMI[7], e.g. (Pazold et al., 2012). An FMI model component exported by a simulation tool is referred to as a Functional Mockup Unit (FMU). An FMU is a zip file containing:

1. The description of the model, e.g. inputs, outputs and parameters in XML format
2. An implementation of the model according to a specific C-API provided either in binary or open-source format
3. Additional optional data and documentation

FMUs can be simulated as standalone applications or a co-simulation slave imported within other simulation tools as model-components, see Figure 3.

[5] `www.modelica.org`
[6] `www.modelisar.com`
[7] `http://www.fmi-standard.org/tools/`

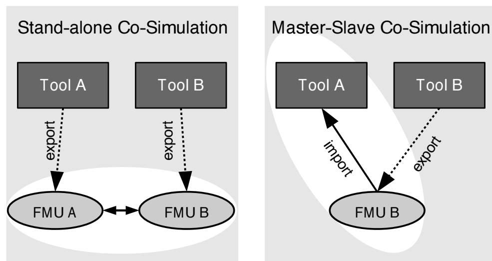

Figure 3: Typical cosimulation scenarios with FMUs. The white area corresponds to the information exchange during co-simualtion.

For the implementation of many FMI-based tools, many open-source existing tools can be used for assisting the implementation, validation and simulation of FMUs, e.g. parsing the XML description file, uncompressing the zip file and accessing its contents. Examples of these tools are FMI library[8], FMI SDK development kit[9], FMU compilance checker[10], PySimulator (Pfeiffer et al., 2012) and JModelica (Andersson et al., 2011). We also provide a high-level C++ library that will be available soon as an open-source for handling FMUs (Widl et al., 2013).

Basic operations

An FMU describes a mathematical model corresponding to a hybrid ordinary differential equation with both continuous and discrete variables as shown in Figure 4 (Elsheikh et al., 2013).

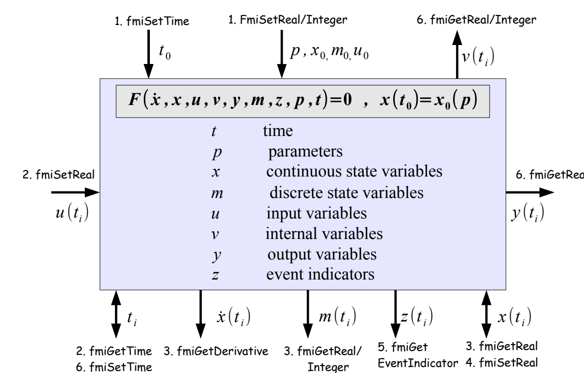

Figure 4: Block diagram of an FMU. Contents of the FMUs are retrieved and updated by FMI function calls. Numerical integration is typically done according to the enumerated FMI operations.

As illustrated in this figure, the numerical integration of an FMU is typically done by the following steps:

1. Initialization step: Setting up start time t_0, model parameters p, start values $x(t_0)$ and $z(t_0)$ and input variables $u(t_0)$.

Then at each time step t_k, the following operations are performed:

2. Preprocessing: Setting the input variables $u(t_k)$
3. Processing: Getting the state variables $x(t_k)$ and the derivative $\dot{x}(t_k)$ and discrete variables $m(t_k)$
4. Integration: Computing $x(t_{k+1})$ from $\dot{x}(t_k)$ using numerical solvers
5. Event handling: Handling the event adequately if an event indicator $z_j(t)$ changes its domain from $z_j > 0$ to $z_j \leq 0$ or vice versa
6. Computing outputs $y(t_{k+1})$ and optionally other intermediate variables $v(t_{k+1})$

Finally, after the last time step is reached:

7. Finalize: Deallocating the memory and process the results

FMI comes in two flavours, FMI for Cosimulation (FMI-CO) and FMI for Model Exchange (FMI-ME). In the former case, the exported FMU includes an integrator, while in the latter case, the developer needs to perform the mentioned steps explicitly. In this work, we make use of FMI-ME and the numerical integration of the FMU is processed by TRNSYS.

THE TRNSYS TOOL

Overview

TRNSYS is a highly-specialized simulation tool capable of modeling and simulating the thermal behaviour of buildings. A graphical editor with high-level functionalities is provided for specifying architectural details and multi-zone structuring. Moreover, a large set of extensively validated model components (TYPEs) like PVs, solar systems, heat pumps and controllers among many others are available. Using the Simulation Studio (SS), the modeler can edit and assemble meaningful models using the provided TYPEs.

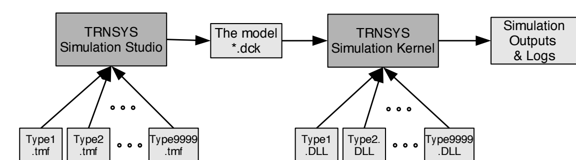

Figure 5: Block diagram of the modeling and simulation process with TRNSYS

TRNSYS provides a modular extensible software architecture for TYPEs implementation and simulation. Each TYPE has a numbered ID. A TYPE implements the physics of a component while its simulation is performed by the simulation engine (the Kernel), see Figure 5. Each TYPE is characterized by an interface and an implementation. The interface specified by the file `Type{k}.tmf` for TYPE number k specifies several quantities: parameters $p \in R^{n_p}$ with default values, inputs $u(t) \in R^{n_u}$, derivatives $\dot{x}(t) \in R^{n_x}$ with default start values $x(t_0)$ and outputs $y(t) \in R^{n_y}$. The modeler can modify the parameter p and start values $x(t_0)$ with the SS. The implementation corresponds to

[8] http://www.jmodelica.org/FMILibrary/
[9] http://www.qtronic.de/en/fmusdk.html
[10] http://www.fmi-standard.org/downloads/

an equation system in the form:

$$\begin{aligned} \dot{x}(t_k) &= f_m(x(t_k), u(t_k), t_k, p) \\ y(t_k) &= g_m(x(t_k), u(t_k), t_k, p) \end{aligned} \tag{1}$$

where $f_m : R^{n_x+n_u+n_p+1} \rightarrow R^{n_x}$ and $g_m : R^{n_x+n_u+n_p+1} \rightarrow R^{n_y}$. A TRNSYS model is assembled by connecting instances of TYPEs together based on the block-diagram approach. In this case, the outputs $y_i(t_k)$ of a TYPE number i become the inputs $u_j(t_k)$ of other TYPE(s) j. This kind of assignments must be consistent. This is ensured by checking that the physical units of the assigned variables are identical. Out of the assembled model, the Simulation Studio generates a specification input file called the Deck, see Figure 5. The Deck is then processed by the Kernel which extracts the following information:

- The parameters p and start values $x(t_0)$
- The present TYPEs and their interrelationships

Then, the Kernel performs the numerical integration.

Implementation of TYPES

All TYPEs are provided as dynamically linked libraries (DLLs) implementing a specific API. Present TYPEs in the Deck are dynamically loaded by the Kernel at run-time. The Kernel communicates with the DLLs and performs the steps shown in Figure 6.

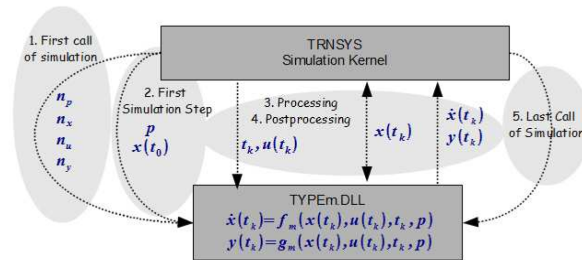

Figure 6: The interaction between the simulation Kernel and a TYPE

The integration steps are summarized as follows:

1. Initialization step: Initializing basic administrative information
2. First call of simulation step: Setting up the parameters p and the start values $x(t_0)$ among other possible operations

Additionally, the Kernel proceeds with the simulation with a given fixed step size Δt. The following operation is performed at each time step t_k:

3. Processing step: computing the derivatives $\dot{x}(t_k)$ (if any) and/or the outputs $y(t_k)$ via the TYPE implementation of Equation (1) using the values $x(t_k)$ and $u(t_k)$ given by the Kernel

Using the computed $\dot{x}(t_k)$, the Kernel performs the numerical integration. In the presence of algebraic loops among components, step 3 is iteratively performed for all TYPEs until some convergence criteria are fulfilled. Afterwards, the following step is performed:

4. Postprocessing step: signalling the convergence of the iterations and performing desired postprocessing operations, e.g. storage of intermediate results, etc.

Finally, after reaching the end of the simulation, a final step is performed:

5. Last call of simulation step: deallocating memory and calling other relevant finalization routines

A TPYE does not need to get numerically integrated by the Kernel. An alternative is to let the TYPE perform the numerical integration itself with its own chosen numerical methods.

FMU-BASED TRNSYS TYPES

The modular architecture of TRNSYS allows inserting self-developed TYPEs implemented in Fortran or C++[11] (Riederer et al., 2009). The implementation should follow a strict template supporting the mentioned operations required by the Kernel. This feature is exploited for providing FMU-based TYPEs for TRNSYS. Adjusting an FMU to a TRNSYS TYPE is straightforward as shown in Figure 7. One of the reasons is that FMI-ME is properly designed for coupling with numerical integrators. The Kernel can be also viewed as a numerical solver of equation systems described by TRNSYS .

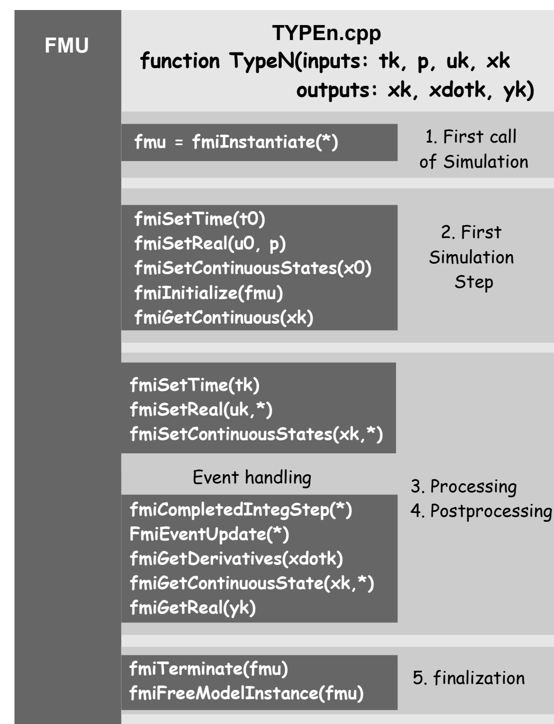

Figure 7: A general pseudo implementation of a template file for importing FMUs into TRNSYS

Following the pseudo implementation in Figure 7, a general C++-template for integrating arbitrary FMUs as TRNSYS TYPEs has been implemented.

[11] theoretically other languages are also possible

TRNSYS-conform FMUs

For enabling such straightforward integration of FMUs, the implementation assumes that the given FMU is TRNSYS-conform satisfying the following conditions:

1. The input-, output- and state variables are explicitly declared within the XML description file
2. Only start values of state variables explicitly declared as model parameters can be modified by a TRNSYS user

Any Modelica model (say `MyModel`) can be easily transformed to a TRNSYS-conform FMU as follows:

Listing 1: Preparing a Modelica model for TRNSYS. Only required entities are present in the public part while the rest of the model is hidden

```
model Type277
  import Modelica.SIunits.*;
  public
    parameter Length p = 0.5;
    parameter Real x2_0 = 1.0;
    ...
    input Real u1;
    ...
    output Volume y1;
    ...
  protected
    MyModel obj(comp1.p = p,
                comp2.x(start=x0));
  equation
    u1 = obj.comp3.u;
    y1 = obj.comp4.y;
    ...
end Type277;
```

In Listing 1, only identities of interest that should be interfaced within TRNSYS are declared within the public part. The reasons for such a transformation are explained as follows:

- Typical FMUs are dimensionally large with so many variables and parameters. Usually, only a smaller subset of parameters and variables are the identities of interest for a TRNSYS user.
- The causality of variables within a Modelica model is not necessarily explicitly declared. In contrary, a TRNSYS TYPE explicitly differentiates between inputs, outputs and state variables, u, y and x, respectively.
- A large number of state variables x would require a lot of efforts from a TRNSYS user for initialization with suitable start values. Therefore, default start values present in an FMU are considered except for start values declared as parameters e.g. `x2_0` in Listing 1.

Note that all these efforts are made only once. Once a TYPE is specified within a *.tmf file and the corresponding *.DLL is successfully compiled, it is available as a TRNSYS type. Moreover, a user does not distinguish between an FMU-based TYPE and normal ones.

EXAMPLE

As a proof of concept, extensive testing has been performed with many Modelica models both from the MSL and self developed abstract ones. The template file was accordingly subject to further improvement on an incremental basis. For any TRNSYS-conform FMU, little manual modification of the C++-template is required for creating an FMU-based TYPE (currently about 4 lines of codes). Nevertheless, a completely automatic process is achievable by employing our high-level FMI++ library (Widl et al., 2013). The FMUs supporting FMI-ME were generated by the Dymola simulation environment. The DLLs were compiled with the gcc 4.7.0 compiler using the MinGW Linux-like environment for Windows. The simulation trajectories are easily comparable with the corresponding simulations with Dymola. So far, no serious differences have been observed at least with the middle-sized tested FMUs.

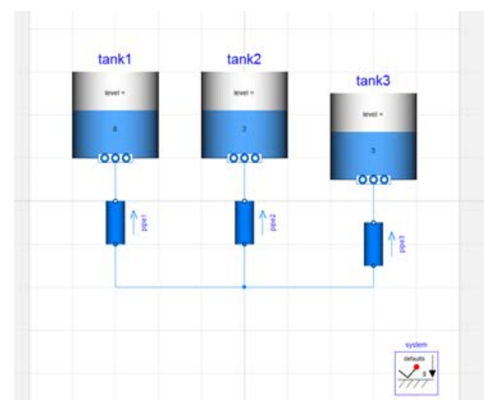

Figure 8: Three tanks positioned at different hight levels connected via pipes with equal diameters

Figure 8 shows a standard model from the `Modelica.Fluid` library for simulating the liquid flow within three tanks. The tanks are placed at different hights and connected with pipes. The compilation of the model results in an equation system with 246 equations. There are 6 state variables and 36 event indicators. Within the corresponding TRNSYS TYPE, the length and diameters of the pipes, the starting hight of the liquids and the hight position of the tanks are parametrized and they can be modified with the SS. Output variables are the dynamics of the liquid volumes and the pressure at the communication ports. Figure 9 shows a corresponding simulation within TRNSYS.

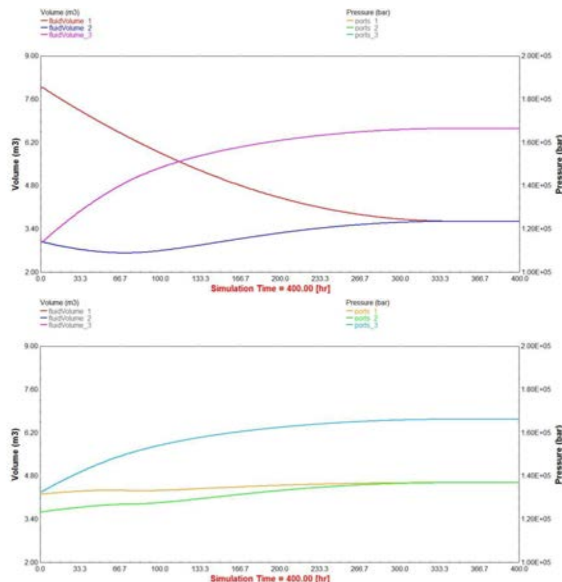

Figure 9: The curves in the upper figure describe the dynamics of the liquid volumes in the three tanks. The left y-axis corresponds to liquid volume. The right y-axis is meaningless.
The curves in the lower figure describe the pressure at the ports of the tanks. The right y-axis corresponds to the pressure while the left y-axis is meaningless.

CONCLUSION AND OUTLOOK

The presented work demonstrates the coupling TRNSYS with Modelica implemented components by providing import functionalities of FMUs within TRNSYS. The imported FMUs are considered as "normal" TRNSYS components and TRNSYS is functioning as the co-simulation master. The presented framework represents a basis for integrating Modelica rapid prototyping capabilities into TRNSYS. Model components implemented with Modelica can be effectively prototyped within a shorter time. Moreover, these components can be easily tested and improved in shorter time. This is particularly interesting for investigating new phenomena and technologies. This is effectively a much faster alternative than low-level implementation of model components with typical procedural languages.

Currently, the presence of multiple FMU instances are not supported by Dymola. Therefore, we aim at integrating FMUs generated with OpenModelica which allows multiple FMUs to be simulated simultaneously. The presented tools are planned to be effectively employed for real life applications joining the advantages of Modelica and TRNSYS. The TRNSYS capabilities for multi-zone modelling will be combined with modified and advanced control strategies implemented in Modelica for energy-efficient buildings design.

REFERENCES

Åkesson, J., Årzén, K.-E., Gäfvert, M., Bergdahl, T., and Tummescheit, H. 2010. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749.

Andersson, C., Åkesson, J., Führera, C., and Gäfvert, M. 2011. Import and export of functional mock-up units in JModelica.org,. In *Modelica'2011: The 8th International Modelica Conference*, Dresden, Germany.

Ashenden, P. J., Peterson, G. D., and Teegarden, D. A. 2003. *The system designers guide to VHDL-AMS*. Morgan Kaufmann.

Åström, K. J., Elmqvist, H., and Mattsson, S. E. 1998. Evolution of continuous-time modeling and simulation. In *ESM'1998: The 12th European Simulation Multiconference - Simulation - Past, Present and Future*, Manchester, United Kingdom.

Bachmann, B., Aronsson, P., and Fritzson, P. 2006. Robust initialization of differential algebraic equations. In *Modelica'2006: The 5th International Modelica Conference*, Vienna, Austria.

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.-V., and Wolf, S. 2011. The functional mockup interface for tool independent exchange of simulation models. In *Modelica'2011: The 8th International Modelica Conference*, Dresden, Germany.

Braun, W., Ochel, L., and Bachmann, B. 2011. Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In *Modelica'2011: The 8th International Modelica Conference*, Dresden, Germany.

Brück, D., Elmqvist, H., Olsson, H., and Mattsson, S. E. 2002. Dymola for multi-engineering modeling and simulation. In *Modelica'2002: The 2nd International Modelica Conference*, Munich, Germany.

Cellier, F. E. 1991. *Continuous System Modeling*. Springer Verlag.

Elmqvist, H. 1978. *A structured model language for large continuous systems*. PhD thesis, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. and Mattsson, S. E. 1997. Modelica - the next generation modeling language: An international design effort. In *ESS97: The 9th European Simulation Symposium*, Passau, Germany.

Elsheikh, A., Awais, M. U., Widl, E., and Palensky, P. 2013. Modelica-enabled rapid prototyping of cyber-physical energy systems via the functional mockup interface. In *The IEEE Workshop on Modeling and Simulation of Cyber-Physical Systems*, Berkeley, USA.

Elsheikh, A., Widl, E., and Palensky, P. 2012. Simulating complex energy systems with Modelica: A primary evaluation. In *DEST'2012: The 6th IEEE International Conference on Digital Ecosystems and Technologies*, Campione d'Italia, Italy.

Franke, R., amd M. Sielemann, F. C., Proelss, K., Otter, M., and Wetter, M. 2009. Standardization of themo-fluid modeling in Modelica.Fluid. In *Modelica'2009: The 7th International Modelica Conference*, Como, Italy.

Fritzson, P. 2003. *Principles of Object-Oriented Modeling and Simulation with Modelica*. Wiley-IEEE Computer Society Pr.

Fritzson, P., Pop, A. D. I., Lundvall, H., Aronsson, P., Nyström, K., Saldamli, L., Broman, D., and Sandholm, A. 2006. OpenModelica - A free open-source environment for system modeling, simulation, and teaching. In *Proceeding of the IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany.

Klein, S. A., Duffie, J. A., and Beckman, W. A. 1976. TRNSYS: A transient simulation program. *ASHRAE Transactions*, 82:623 – 633.

Maffezzoni, C., Girelli, R., and Lluka, P. 1996. Generating efficient computational procedures from declarative models. *Simulation Practice and Theory*.

Mattsson, S. E. and Elmqvist, H. 1997. Modelica - an international effort to design the next generation modeling language. In *CACSD'97: The 7th IFAC Symposium on Computer Aided Control Systems Design*, Gent, Belgium.

Pantelides, C. C. 1988. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231.

Pazold, M., Burhenne, S., Radon, J., and amd F. Antretter, S. H. 2012. Integration of modelica models into an existing simulation software using fmi for co-simulation. In *Modelica'2012: The 9th International Modelica Conference*, Munich, Germany.

Pfeiffer, A., Hellerer, M., Hartweg, S., Otter, M., and Reiner, M. 2012. PySimulator – A simulation and analysis environment in Python with plugin infrastructure. In *Modelica'2012: The 9th International Modelica Conference*, Munich, Germany.

Riederer, P., Keilholz, W., and Ducreux, V. 2009. Coupling of TRNSYS with Simulink – A method to automatically export and use TRNSYS models within Simulink and vice versa. In *Building Simulation 2009, The 11th international IBPSA Conference*, Glasgow, Scotland.

Wetter, M. 2009. Modelica-based modelling and simulation to support research and development in building energy and control systems. *Journal of Building Performance Simulation*, 2:143 – 161.

Wetter, M. and Haugstetter, C. 2006. Modelica versus trnsys – a comparison between an equation-based and a procedural modeling language for building energy simulation. In *The 2nd SimBuild Conference*, Cambridge, MA, USA.

Widl, E., Müller, W., Elsheikh, A., Hörtenhuber, M., and Palensky, P. 2013. The FMI++ library: A high-level utility package for FMI for model exchange. In *The IEEE Workshop on Modeling and Simulation of Cyber-Physical Energy Systems*, Berkeley, USA.