Visualizing Simulation Results from Modelica Fluid Models Using Graph Drawing in Python

Marcus Fuchs Rita Streblow Dirk Müller

RWTH Aachen University, E.ON Energy Research Center, Institute for Energy Efficient Buildings and Indoor Climate, Aachen, Germany, mfuchs@eonerc.rwth-aachen.de

Abstract

Models of large thermo-fluid networks can be useful to better understand the dynamic behavior of complex systems. Yet, numerical outputs and line plots of individual variables may not be sufficient ways of processing the simulation results for the user. Thus, the aim of this paper is to present a visualization approach by means of graph drawing. To demonstrate the approach, we use an example from the Modelica Standard Library and the use case of a district heating system model. We parse the Modelica model code to generate a `System` graph that represents the model structure and its graphical layout. The graph drawing subsequently visualizes the results for every time-step. In the examples, we vary line thickness to visualize mass flow rates between two nodes and line color to show temperatures of the medium. We argue, that this approach can be a useful tool for modeling and analysis.

Keywords: Visualization, Graph Drawing, Modelica Fluid, District Energy System

1 Introduction

One reason for using the Modelica modeling language is the high re-usability of component models from model libraries. In this context, the acausal connections between component models can be used to efficiently assemble larger system models (Dizqah et al., 2015). For thermo-fluid systems, the `Modelica.Fluid` (Casella et al., 2006) package includes the concept of stream connectors, which facilitates the modeling of flow networks with possible flow-reversals. In energy systems modeling, e.g. for building or district heating systems, this enables the assembly of large system models from only a limited number of component models like pumps and pipes.

When connecting multiple `Modelica.Fluid` component models in a pipe network, the fluid flow is driven by pressure differences between connectors. Often, models provide a relationship between mass flow rate and the pressure drop between the component's ports. This leads to a network of mass flows between different pressure levels. In many cases, another key aspect of modeling are the thermal properties of the fluid flow and parts of the components. A system model containing information about all these aspects can be very useful to understand the system's dynamic behavior. Yet, with increasing system size this amount of data increases at a rate that can make it hard to comprehend and verify simulation results. In these cases, numerical outputs and line plots of individual variables may not be sufficient ways of processing simulation results for the user. Thus, the aim of this paper is to present an approach to visualize the information from thermo-fluid system simulations by means of graph drawing and animation.

The need for additional visualization approaches when dealing with complex Modelica system models and its advantages for the user's understanding has been highlighted before. Previous work on this topic has mainly focused on 3D visualization. To this end, Höger et al. (2012) present an approach called `Modelica3D`, in which Modelica code is used to communicate with 3D rendering tools. They show the applicability of this approach for multi-body systems as well as in a building energy system context, with a focus on the 3D visualization of each component. In addition, Hellerer et al. (2014) give a wide range of examples for their `DLR Visualization Library` with a focus on multi-body simulations. Both these papers also give a similar overview of other previous work on this topic. Furthermore, simulation environments like Dymola offer functionalities for plotting and animations of 3D objects, also with a focus on multi-body animations.

In addition to the focus on multi-body and 3D visualization, the field of thermo-fluid modeling has also in part relied on post-processing simulation results using the programming language Python. As a result, there are several Python packages with different functionalities available. One such package is `BuildingsPy` (LBL-SRG, 2015), which among other functionalities contains methods for managing simulations, unit testing model libraries, and processing result files. The package `awesim` (De Conick, 2015) is a tool that helps to manage a variety of simulations and result files and so is use-

ful for multiple simulations and parameter studies. In addition, the package ModelicaRes (Davies, 2015) provides a user-friendly approach to process and plot simulation results. There are thus various approaches to read and work with Modelica simulation results in Python.

When considering the processing of simulation results for thermo-fluid networks, a promising method to represent the model structure is in graph format. In a non-Modelica context, e.g. Fang and Lahdelma (2014) use a graph notation to describe a district heating network with pipe elements as edges connecting the network nodes. One useful approach to use such graph notation in Python is to use the package networkX Hagberg et al. (2008). networkX is a free package providing data structures and algorithms for different graph types and work involving different kinds of complex networks. Furthermore, its open design allows for a wide variety of data to be represented by nodes and edges. Together with the powerful plotting package matplotlib (Hunter, 2007), networkX can be used to visualize graphs in many ways.

Building on the previous work done by the Python developers mentioned above, we set out to present an approach for visualizing the dynamic behavior of complex thermo-fluid networks modeled in Modelica by means of a Python post-processing.

2 Process Overview

The approach presented in this paper aims at producing a visual output, helping to better comprehend and analyze the data produced by simulating complex thermo-fluid networks in Modelica. To this end, we use a post-processing routine in Python. Python was chosen as a programming language, in part because of its accessibility through easy syntax, wide use, and being platform-independent. Another advantage of using Python is the possibility to build on the previous work done in post-processing Modelica results as described in section 1.

Fig. 1 shows a schematic representation of the approach presented in this paper. The information contained in a Modelica model is used to initialize a Model graph object. As it is often helpful to make abstractions from the original model design for visualization, this Model graph is transformed to a System graph in a subsequent step. After reading the Modelica model's simulation results to the System graph, this class can generate a visual output in the form of static plots and video animations. Both the python classes for the Model and the System extend the class nx.Graph from networkX, so that it inherently has all of networkX's well established functionalities for graph handling and analysis. In order to read the Modelica result files and process the data, the code uses the ModelicaRes package. This way, the Model and System classes can be focused on performing the visualization without the overhead of reproducing graph and result handling functionalities already available elsewhere.

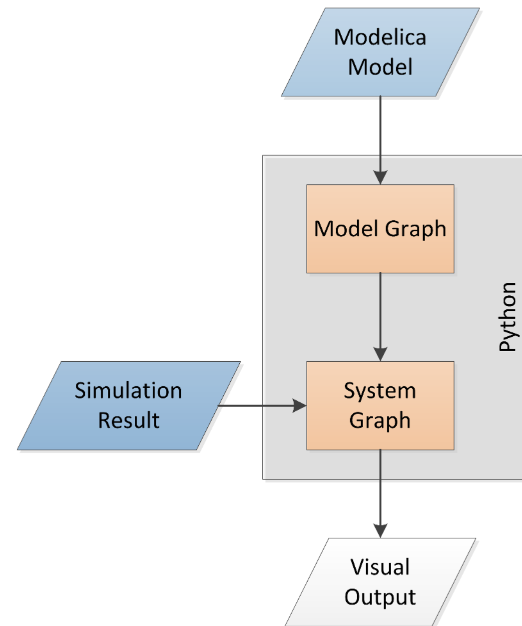

Figure 1. Flow chart for creation of visual output from Modelica model

Reading information from a Modelica model to the Model class in Python marks the start for the described process. This information is represented in the graph by placing edges between the nodes. In order to arrive at a more intuitive display of the model structure, especially for complex pipe networks, the Model graph is transformed to a System graph. One major change in that transformation is the introduction of network nodes between sub-models. In a further step, pipe models are transformed from individual nodes to edges connecting the network nodes. With pipes serving as connecting elements in real-world systems, this representation may be more user-friendly for the following visualization. The process is described in more detail in section 3.

As a second input to the visualization process, the System class uses methods from ModelicaRes to read data from the result file into Python. This data can be selected according to the purpose of the visualization. Yet, for analyzing thermo-fluid systems, we will concentrate on the processing of mass flow rates, pressures, enthalpies, and temperatures. In order to handle this data efficiently, networkX allows to attach almost any kind of data and objects to individual nodes and edges. Thus, each node and edge representing a model component can hold its relevant information from the Modelica result file. As the dynamic behavior of the system is of special interest, each dataset contains the time-series of data for every time-step of the simulation.

The data attached to nodes and edges can subsequently be used to visualize the overall system behavior

DOI
10.3384/ecp15118737

by means of graph drawing. In addition to static graph drawings showing the graph's structure, the information contained in the graph drawing can be extended by different means. For this demonstration, we will use the line thickness and color of edge connections to represent mass flow rates and temperatures for every simulation time-step. In section 7 we will point to further possibilities of enriching this data visualization approach in future work.

After visualizing the system properties for every time-step, we create a video from the individual plots, which as a final outcome produces an accessible and intuitive way to animate an amount of data for a complex system that would be hard to process for a human user in a standard 2D line plot. In the following sections, we will present the individual steps outlined above in more detail for an example model from the Modelica Standard Library. After that, we will present a use case of a campus-scale district heating network to demonstrate the capabilities of the visualization approach in an applied context.

3 Translation of Modelica Model to Graph

As outlined above, the developed `Model` class aims at a representation of the Modelica model in a graph structure using Python. We will use the model `IncompressibleFluidNetwork` from the Modelica Standard Library's `Modelica.Fluid.Examples` package to illustrate the process description. The model's diagram view is shown in Fig. 2. This system consists of a piping network with 11 pipes and 3 valves, transporting fluid flows from a source on the figure's left side to a sink on the figure's right side. For reasons of clarity, we will limit the processing of this example to basic functions. The full capability of the presented approach in its current state will be shown in section 6 for the example of a district heating network model.

For the first processing step, the `Model` class includes methods to parse the Modelica code of a given file and extract information from its declaration sections as well as from the equation section. These functionalities are of limited scope, however, as they focus only on mapping the model structure into a graph in Python. More complex Modelica features such as extending and redeclaring are not processed by this simple parser. For the component model declarations, the parser extracts data about the component's class, its instance name as well as the coordinates of its graphical representation, which can be read from the corresponding annotation. At the current stage, this step will process only declarations of components that have been selected in advance. This limitation arises from the fact that later in the process, special subclasses are needed to extract relevant information from the simulation results for each type of component.

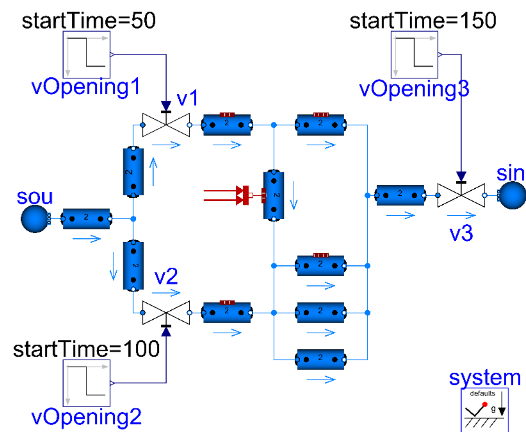

Figure 2. Diagram view of the example model for an incompressible fluid network from `Modelica.Fluid`

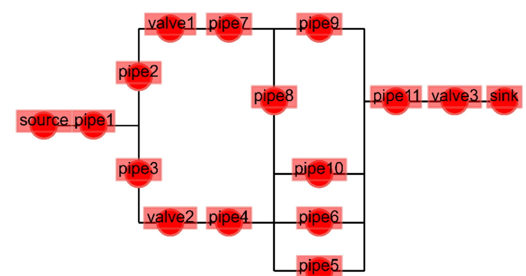

Figure 3. Representation of the example model in a `Model` graph

In the example model, we prepared only for the fluid components to be processed. As a result, sub-models that are not an integral part of the fluid network, like the system model in the lower right corner of Fig. 2 and the control inputs for the valve openings are not taken into account to be part of the `Model` graph. If of special interest, a processing of these sub-models could also be implemented into the presented framework. Yet, for larger fluid networks, this may compromise the clarity of the visualization.

Having set all the graph's nodes according to the relevant model components, the parser returns to the Modelica model file to extract the connection statements. For each connection statement involving those component instances that are represented as a node, an edge is added to the graph accordingly. In order to conserve all relevant graphical information from the Modelica code, the parser also processes the annotations of the connection statements. If a connection in the Modelica code is not drawn directly between two ports as a straight line, the intermediate points given in the annotations are inserted to the graph as separate network nodes. As a result, the graphical representation of the graph will better match the original Modelica model. For the example model shown in Fig. 2, the `Model` graph is displayed in Fig. 3.

Before reading simulation data to the graph, we sug-

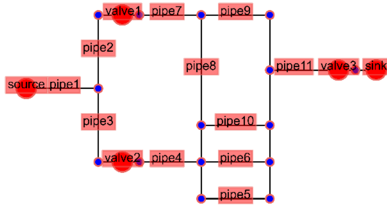

Figure 4. Representation of the example model in a `System` graph. Network nodes are displayed in blue

gest converting the `Model` graph to a `System` graph. This transformation can help to make the visualization more intuitively comprehensible to the user. In this example, we transform the `Model` graph in such a way that the pipes are converted to be edges between network nodes instead of nodes themselves. For this example, we decided to keep the valves as nodes, thus showing both possible pathways of keeping a component type as nodes and converting nodes to edges for the pipes. This could be changed according to the specific application with little effort.

Fig. 4 shows the result of the conversion, with the network nodes marked in a blue color and the pipes being represented by edges. Even though the advantages of this transformation may not be highly significant for this simple example, the use case in section 6 will demonstrate the benefits in the context of a larger pipe network. Furthermore, the distinction between `Model` and `System` graphs allows for more dedicated class definitions with focus on parsing the Modelica file for the `Model` class and focus on visualization for the `System` class.

4 Reading Result Data to Graph

The `System` graph is created as a data structure and template to visualize the dynamic system behavior. In order to read the simulation result data into this structure, the `System` class can access top-level system data directly by making use of result handling methods from the package `ModelicaRes`. For handling result data of the individual components, `System` calls special `Component` classes. A basic `Component` class defines methods for extracting certain data from the result file for a component in a general way. Examples for such methods are `get_mass_flow_rate` or `get_temperature`, which return the time-series of mass flow rate or temperature in the component respectively.

As the identifiers for each component's variables may be different, we extend this general `Component` class for every relevant component type and assign it its own class with specific identifiers and in some cases with special functionalities. In the example of 2, three such classes are needed, i.e. the classes `Boundary`, `Valve`, and `Pipe`. As extending the `Component` class requires relatively little effort, we prefer this method over the attempt to have only one `Component` class that tries to manage all different component types and their differences.

In terms of processing the data, when adding a node to the `Model` graph, a `Component` object is automatically initialized and attached to the respective node. To this end, `networkX` allows for setting data and objects as attributes to nodes, edges, or the graph itself. In the implemented approach, each object that is attributed to a component's graph representation is also moving from the `Model` into the `System` graph. As a result, all data regarding the component can be accessed by user-friendly methods like the `get_mass_flow_rate` mentioned above for each node and edge. Thus, the object-oriented approach from the Modelica model is followed also in the post-processing by an object-oriented Python implementation.

For components that do not correspond to any Modelica component directly, in some cases the code will assign the object of a neighboring graph element. For example a network node may thus be attributed an instance of the neighboring pipe object, so that it will return the pipe's mass flow rate when queried for such data. In some cases, like for an edge between two network nodes, there may also not be a neighboring object that directly represents a Modelica model component. For this situation, we sometimes prefer not to attach any data to it in order to not give any wrong impression in the visualization. Yet, there is the possibility to interpolate some of this data when the user wants it visualized in a certain way.

When thinking about ways to visualize different kinds of data for various components, some ways of display seem more intuitive than others. As mentioned above, some of the most relevant data to visualize for a thermo-fluid network are mass flow rates, pressures, enthalpies, and temperatures. Often, mass flow rates and temperatures are of special interest. In a non-Modelica context, Köcher (2000) reported a way of visualizing pressures and temperatures in a district heating network at the nodes. Yet, in the `System` class representation shown in Fig. 4, most of the mentioned information concerns the edges rather than the nodes. Thus, the way the edges are drawn in a graph plot are a central part of the visualization. In order to prepare that visualization, we use selected data to calculate edge weights and edge colors to be used in the plotting.

The selection of what values to represent by edge weights and colors is up to the user. The `System` class contains methods for both calculations, that take as arguments the variable that is to be represented, e.g. `temperature` or `mass flow rate`. Based on this selection, the edge weight and color will be calculated and attributed to the corresponding edge. In the case of color representation, a relative value between the mini-

DOI 10.3384/ecp15118737

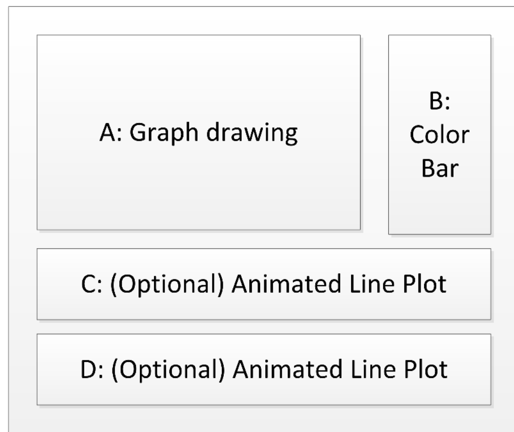

Figure 5. Schematic view of the visualization grid structure

mum and maximum value will be calculated and mapped to a color coding using `matplotlib`'s color-mapping function.

After the graph construction, transformation, and result file handling, the `System` graph will contain all relevant data for visualization. This data structure concept has proven to be user-friendly and efficient, making all data easily accessible and fast to process. The `System` representation as described above presents a compromise between the most intuitive design and strictly following the Modelica model setup. This compromise may be evaluated for each use case and the graph representation adjusted accordingly. This is possible with moderate manual effort, as the object-oriented and graph-based code structure should be reasonably transparent for the user.

5 Visualization of Fluid Flows

In order to keep the visualization output as flexible as possible, we define a framework for sub-plots using `matplotlib`'s grid structure. Fig. 5 illustrates the concept. The only fixed properties are the spaces *A* and *B* that serve as placeholders for the network graph drawing and the corresponding color map. In many cases, one can argue that such a graph drawing has advantages over a multitude of standard 2D line plots. Yet, we do not want to argue that it is inherently always superior to the clarity and simplicity of a line plot. Therefore, any number of line plots can be placed beneath the graph drawing in any number of spaces *C*, *D*, and so on. The `System` class allows the user to name the variables that should be plotted in addition to the graph drawing.

Regarding the graph drawing for space *A* in Fig. 5, the user can select different visualization types. Most times, this will consist of a 2D view recreating the `System` graph as illustrated in Fig. 4, with the edge weights and colors varying according to the preselected variables. For the future, we will also work on 3D plots, where the value of an additional variable can be visualized by use of a z-axis. This is especially interesting to visualize pressure levels so that mass flows will flow from nodes plotted at greater z-axis levels to those with lesser z-values.

In any case, the Python routine will create a plot following the structure shown in Fig. 5 for every time-step in the simulation result file or for a user-selected period within the simulation time limits. The graph drawing will loop over all nodes and edges, plotting them into space *A* and adjusting their appearance according to data like the node type, edge weight, and edge color stored in the `networkX` graph data structure. For the line plots, the lines will be drawn from the time-step at the beginning of the visualization until the current time-step for this plot. Thus, when the individual static plots are compiled into a video, this will give the impression of a line plot tracking the behavior of the corresponding variable with each time-step.

Returning to the illustrative example introduced with Fig. 2, we can demonstrate the graph drawing part of the visualization output. Unfortunately, as the presented approach directly aims at overcoming the limits of static data plotting, it is hardly possible to show the benefits of an animated visualization in the form of this paper. Therefore, we attempt to mitigate this shortcoming in the paper by using the timeline representation given in Fig. 6. In order to avoid distractions, we limited the display to the plain graph drawing for four steps during the simulation time of 200 s.

There are three changes happening during simulation, namely the closing of valve 1 after t = 50 s, the partial closing of valve 2 at t= 100 s and the partial closing of valve 3 after t = 150 s. The graph drawings show how these changes affect system behavior. After the closing of valve 1, the upper pipe branch is cut off from the flow between the source at left and the sink at right. The partial closing of valves 2 and 3 shows the effect of a reduced mass flow rate in the whole system, depicted by a thinner line thickness for all connections.

In order to better demonstrate the functionalities of the color mapping, we made one slight change to the original model from `Modelica.Fluid`. In the original model, the temperature at the source is kept constant, and the heat source in pipe 8 only has a limited effect on the system as a whole. Therefore, we changed the source temperature to start at 80 °C and decrease linearly until the end of the simulation to 20 °C. This decreasing temperature can be seen in Fig. 6, represented by the changing edge colors. In calculating the edge colors, we used temperature values derived from the average enthalpy between the two fluid ports of a component.

For animating the individual plots in a video, we use the lightweight and freeware software `Images to Video` (Sivic, 2015). This software can be called via a command line interface with all settings saved in an XML file. As these steps can be executed from within the Python environment, the solution requires no effort

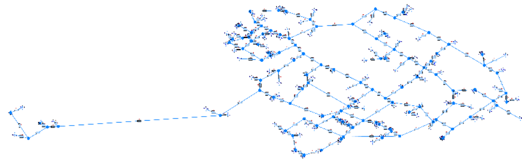

Figure 7. Diagram view of the district heating network model

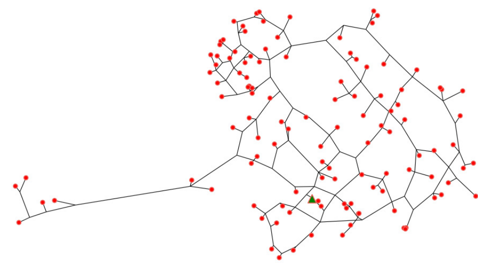

Figure 8. The district heating network's `System` graph

from the user. Also, the user is free to work directly with the individual plots created or use different software to create a video. Still, this part could be improved upon if a Python package for creating the video could be used instead for a more integrated process.

6 Use Case: District Heating Network

In the previous sections, we used a rather academic example to demonstrate the process and functionalities of the presented approach. In this section, we show a use case for which the presented visualization tools were originally developed. We investigate a district heating network that supplies about 120 buildings with heat from one central heating plant. To model this system, we use simplified component models for pipes, the building substations, and the supply. The graphical representation of the system model is shown in Fig. 7. The pipe models calculate a pressure drop depending on the mass flow rate and have a thermal connection to the ground temperature to calculate thermal losses. The building substation models include a control valve, adjusting the mass flow rate according to building heat demand given as a table input. The supply model consists of a simple pump model and an ideal heat source, controlling the network's supply temperature to a set temperature depending on the outdoor air temperature.

Considering the about 120 buildings, over 200 pipe elements in the supply and return lines, and the loops in both, the district heating network qualifies as a complex thermo-fluid system. Using only 2D line plots to visualize mass flow rates and temperatures for the entire system can thus be cumbersome. In this context, the presented visualization approach can help to verify and better understand the system behavior of the model.

For the system model, there are two largely identical pipe networks, one for the supply lines from supply plant to the buildings and one for the return lines from buildings to the supply plant. We modeled both these networks, but only used graphical annotations for the Modelica code of the supply lines. Therefore, the return pipes and their connections are not shown in the diagram view of the Modelica model. This leads to a clearer model view, yet makes it even more important to verify the model results in order to ensure that all these connections are correct.

As the `Model` class processes the Modelica code in terms of declaration statements, connections, and their graphical annotations, the missing graphical annotations lead to the return components neither being represented in the `Model` nor in the `System` graph. The resulting `System` graph for the district heating network is shown in Fig. 8. Nevertheless, the values of the return pipes can be shown in the graph in place of the supply pipes' values, as the return lines are placed at the same locations as the supply lines. For the data handling of each component, we extend the general `Component` class and

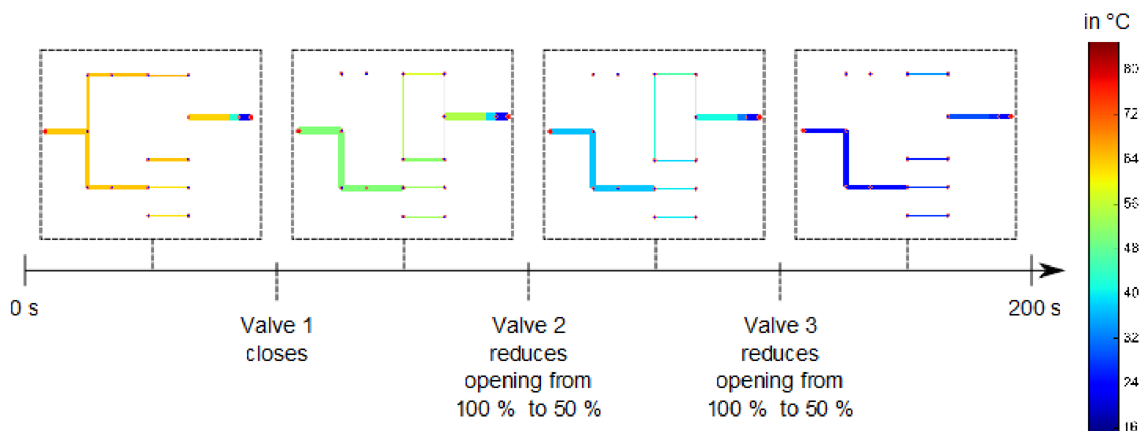

Figure 6. Using graph drawing to visualize system behavior over time

DOI
10.3384/ecp15118737

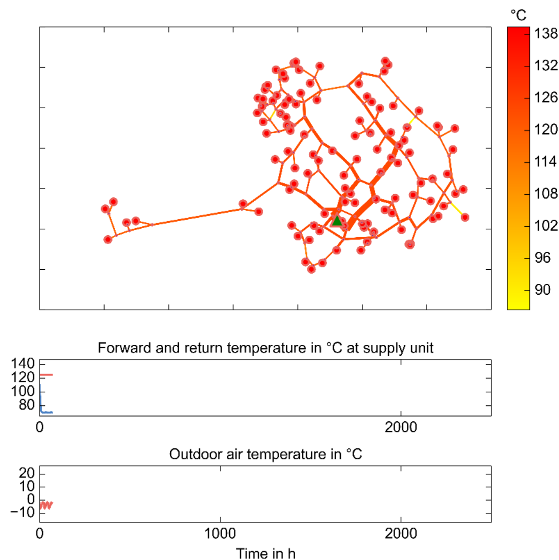

Figure 9. Visualization output for the district heating network at a low-load operation at the beginning of the simulation

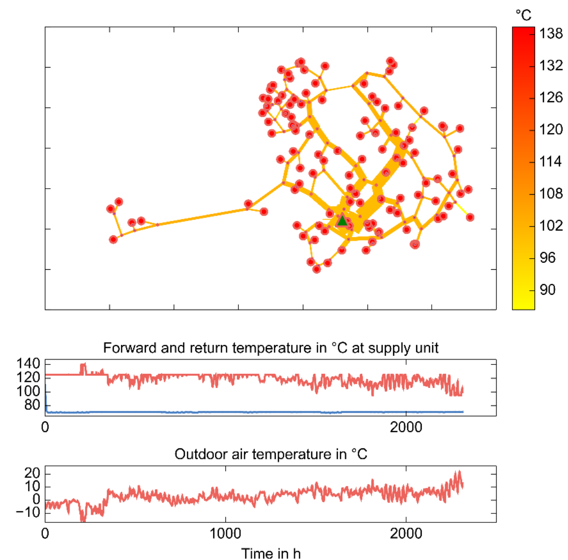

Figure 10. Visualization output for the district heating network at a medium-load operation near the end of the simulation

define the identifiers for different variables in the result file as described in section 3. Thus, it is possible to use a `Pipe` component class that retrieves the supply or the return pipes' data at the user's selection. Similarly, we use a `Supply` and a `Building` class to handle the data of these components.

To demonstrate the visualization approach for this use case, we use a simulation of the district heating network model for a simulation time of 2500 hours with an hourly time-step. Starting at the beginning of the year, this illustrates the first part of the year with significant heat loads. Following the layout of Fig. 5, we use the graph drawing to visualize mass flow rates and the temperatures in the supply lines as well as two line plots. One line plot shows the supply plant's supply and return temperatures and the second line plot shows the ambient outdoor temperature as read from the weather input file.

Again, this paper is limited to show static plots of the visualization for given time-steps. A further application is to compile a video from all the plots to animate the dynamic system behavior. For this demonstration, Fig. 9 shows the state of the system near the beginning of the simulation while Fig. 10 shows the system closer to the end of the simulation. By varying the line width of the pipe connections according to pre-calculated edge weights that depend on the mass flow rate, it is possible to show the mass flow rates for all of the supply lines' over 100 pipe elements in one single plot. Together with the color mapping of water temperatures within the pipe to the color bar given on the upper right, the plots give an impression of the energy flows in the network.

A comparison of Fig. 9 and Fig. 10 illustrates the concept of line plotting in the lower part of the figures. As the line is plotted from the beginning until the current time-step, it gives an impression of monitoring the selected simulation results when animated into a video. Furthermore, it serves as an indicator of the current time-step even in a static plot and enables the plotting of data that would be difficult to represent by color and line thickness or variables that are not directly part of the thermo-fluid network like the ambient temperature. This enriches the context for the graph drawing visualizing the energy flows in the network.

By visualizing energy flows in the graph drawing, the presented approach can be a useful tool in verifying simulation results. For verification purposes, the main advantage of the graph-drawing based visualization approach over simple line plots is that various system variables are shown together and in context of the system behavior. Furthermore, it would be possible to not only display simulation result variables in such a visualization, but to also display deviations from measurement data, if such data is available.

Once verified, the visualization can also be used as a tool to better understand system behavior and thus assist in planning of the system operation. In real-world thermo-fluid networks, the exact ways the energy flows take is often not known. Especially in district heating networks that include multiple loops and where the pipes are buried in the ground, it can be hard to measure the direction and flow rates of all the pipes. In these cases, as Fig. 10 indicates, the visualization can help to identify main routes of energy flows as well as pipe elements with low flow rates. Yet, to draw conclusions for the operation of the actual system, efforts must be made to verify such observations in the real-world system, as model assumptions and malfuntions in the actual system can lead to deviations between simulation and real-world operation.

7 Conclusions

This paper presents an approach that uses post-processing of Modelica simulation results and graph drawing in order to better visualize the dynamic behavior of complex thermo-fluid networks than standard line plots of individual result variables. Using a graph and attributes for nodes and edges as a data-structure to handle Modelica simulation results has proven a feasible concept, as it can mirror the object-oriented structure of the Modelica model into the post-processing. This allows for a low-maintenance framework that nevertheless offers flexibility for adjustments and options to tailor the visualization output to the specific aims of the visualization and to the requirements of the used models.

Regarding the computational performance, processing the data as well as the `Model` and `System` graphs creates little overhead and takes a few seconds on a standard laptop computer. The time for the plotting will largely depend on the model size, time-step, simulation time, and the required resolution of the output data. Therefore, this part of the process can currently take from a few minutes up to 2 hours for a very high-resolution animation of a large district heating system simulation with small time-steps and a duration of 1 year. Yet, it is likely that the time this part of the process can be efficiently reduced by parallelization of the plotting.

The functionality of the presented approach was demonstrated for a simple example from the Modelica Standard Library as well as for a real-world application of a district heating system model. In this proof of concept, we used line thickness to visualize mass flow rates from one node to another and line colors to indicate temperature levels. Other possible uses include visualizing pipe diameters with line thickness or flow velocities with line colors. Also, we limited our graph drawing to 2-dimensional representations of the system, which resembles the diagram view of the corresponding Modelica models. In this process, parsing the Modelica code for the graphical information in the annotations leads to nodes in the graph with corresponding coordinates. In future work, it will be interesting to visualize certain values in a pseudo-3-dimensional way, where the model representation can stay in the x- and y-axes while simulation result values can be shown on a corresponding z-axis. This is especially promising to visualize pressure levels of supply and return lines for thermo-fluid networks or deviations between simulation results and measurement data.

We argue that the presented approach can be a useful tool in handling the complexity of larger thermo-fluid networks and their dynamic system behavior. On the one hand, the visualization of energy flows and other simulation result data can help modelers to verify their model setups and assumptions. On the other hand, the visualization can be used to inform about relationships and interactions of system components. Yet, drawing conclusions from such visualization for the operation and design of actual systems, similar to all aspects of modeling and simulation, requires critical verification of the models used and the results obtained.

Furthermore, the process of visualizing Modelica simulation results introduces methods to parse Modelica code and handle information about model structure and behavior in a Python-based graph structure. For the future, it will be interesting to use these resources for the automated generation and modification of Modelica models. To this end, we are working on a bi-directional work-flow to generate Modelica models for district energy systems from different input data with the `System` graph at the conceptional core. Possible input data includes data from geographic information systems (GIS) or CityGML. In reverse, these models and their results can again be processed by the `System` graph as described in this paper. Thus, the `System` graph can be used as the foundation in an integrated workflow for model generation as well as result analysis and visualization. We think that such an approach has the potential to address handling the complexity of input and output data of large-scale energy system models, which has been identified as one of the key challenges in modeling such systems (Keirstead et al., 2012).

This will hopefully reduce manual effort in modeling complex system like district energy systems and lead to insights from modeling these systems for real-world applications. To this end, we plan to release the developed Python code as an open-source package in the near future. In addition, the Modelica component models for the district heating network modeling will be made available through the open source model libraries `AixLib`[1] and its contributions to the Annex 60 library[2], which is a joint effort within the International Energy Agency's Annex 60 programme.

Acknowledgment

We gratefully acknowledge the financial support by BMWi (German Federal Ministry of Economic Affairs and Energy), promotional reference 03ET1260A.

References

Francesco Casella, Martin Otter, Katrin Proelss, Christoph Richter, and Hubertus Tummescheit. The Modelica Fluid and Media library for modeling of incompressible and compressible thermo-fluid pipe networks. In Modelica Association, editor, *Proceedings of the 5th International Modelica Conference*, pages 631–640, 2006.

Kevin Davies. ModelicaRes python package, 2015. URL `http://kdavies4.github.io/ModelicaRes/`.

[1] httpd://github.com/RWTH-EBC/AixLib
[2] https://github.com/iea-annex60/modelica-annex60

DOI 10.3384/ecp15118737

Roel De Conick. awesim python package, 2015. URL https://github.com/saroele/awesim.

Arash M. Dizqah, Alireza Maheri, Krishna Busawon, and Peter Fritzson. Standalone DC microgrids as complementarity dynamical systems: Modeling and applications. *Control Engineering Practice*, 35:102–112, 2015. ISSN 09670661. doi:10.1016/j.conengprac.2014.10.006.

Tingting Fang and Risto Lahdelma. State estimation of district heating network based on customer measurements. *Applied Thermal Engineering*, 73(1):1211–1221, 2014. ISSN 13594311. doi:10.1016/j.applthermaleng.2014.09.003.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

Matthias Hellerer, Tobias Bellmann, and Florian Schlegel. The DLR Visualization Library - recent development and applications. In *the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*, Linköping Electronic Conference Proceedings, pages 899–911. Linköping University Electronic Press, 2014. doi:10.3384/ECP14096899.

Christoph Höger, Alexandra Mehlhase, Christoph Nytsch-Geussen, Karsten Isakovic, and Rick Kubiak. Modelica3D - platform independent simulation visualization. In Modelica Association, editor, *Proceedings of the 9th International Modelica Conference*, pages 485–494, 2012. doi:10.3384/ecp12076485.

John D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

James Keirstead, Mark Jennings, and Aruna Sivakumar. A review of urban energy system models: Approaches, challenges and opportunities. *Renewable and Sustainable Energy Reviews*, 16(6):3847–3866, 2012. doi:10.1016/j.rser.2012.02.047.

Ralf Köcher. *Beitrag zur Berechnung und Auslegung von Fernwärmenetzen*. PhD thesis, Technische Universität Berlin, Berlin, 2000. URL http://d-nb.info/960177469/34.

LBL-SRG. BuildingsPy python package, 2015. URL https://github.com/lbl-srg/BuildingsPy.

Jaromir Sivic. Images to video v4.0, 2015. URL http://en.cze.cz/Images-to-video.