OpenModelica

a Quick Start Guide

B. MASEFIELD
SOLVEERING LLC
OCTOBER 2019

Applicable to OpenModelica V1.14.0 (dev)

Introduction

This quick-start guide is meant to provide a short introduction to the OpenModelica Application, specifically to the setup and use of fluid (and similar) systems using OMEdit. At this time, there are few complete manuals available that aim at providing a basic overview of the editor and how it relates to getting a simple system to run. This document is by no means intended to substitute the materials available in more detailed courses [1], application documentation [2], [3] or other reference material [4] . Instead its primary focus is to allow someone -who has a basic understanding of the physics and engineering aspects but no prior knowledge of the application or the processes by which the application provides solutions- to quickly set up a simple system and solve it or load someone else's work and modify/update some basic components or values.
For information on the Modelica language [3], download or other materials, the reader is provided with a short (and incomplete) list in the references section.

Installation

The application can be downloaded from [5]. There exists different installation media (Windows, Mac, Linux, VM) and for each there are stable and development versions. This document is based on V1.14.0 which is at the time of this writing a development-only version. Despite a number of minor bugs (related to graphics and the occasional crashes), the 1.14 version generally runs better.

After downloading the application (The v1.14.0 latest/nightly build version is approximately 1.3GB in size), installation is accomplished using the normal method (running as administrator). One item of importance is that the application is installed in a directory without special characters (so the default of "C:/Program Files/OpenModelica/" is not recommended due to the space).

Upon completing the installation, a number of Tutorials, Guides and application short-cut items will be installed under the OpenModelica folder (or similar depending on installation options/other operating systems). The shortcut of interest is that to the OpenModelica Connection Editor (OMEdit):

OMEdit (the GUI)

Upon starting the OpenModelica Connection Editor (OMEdit) application, the default view is shown in Figure 1:

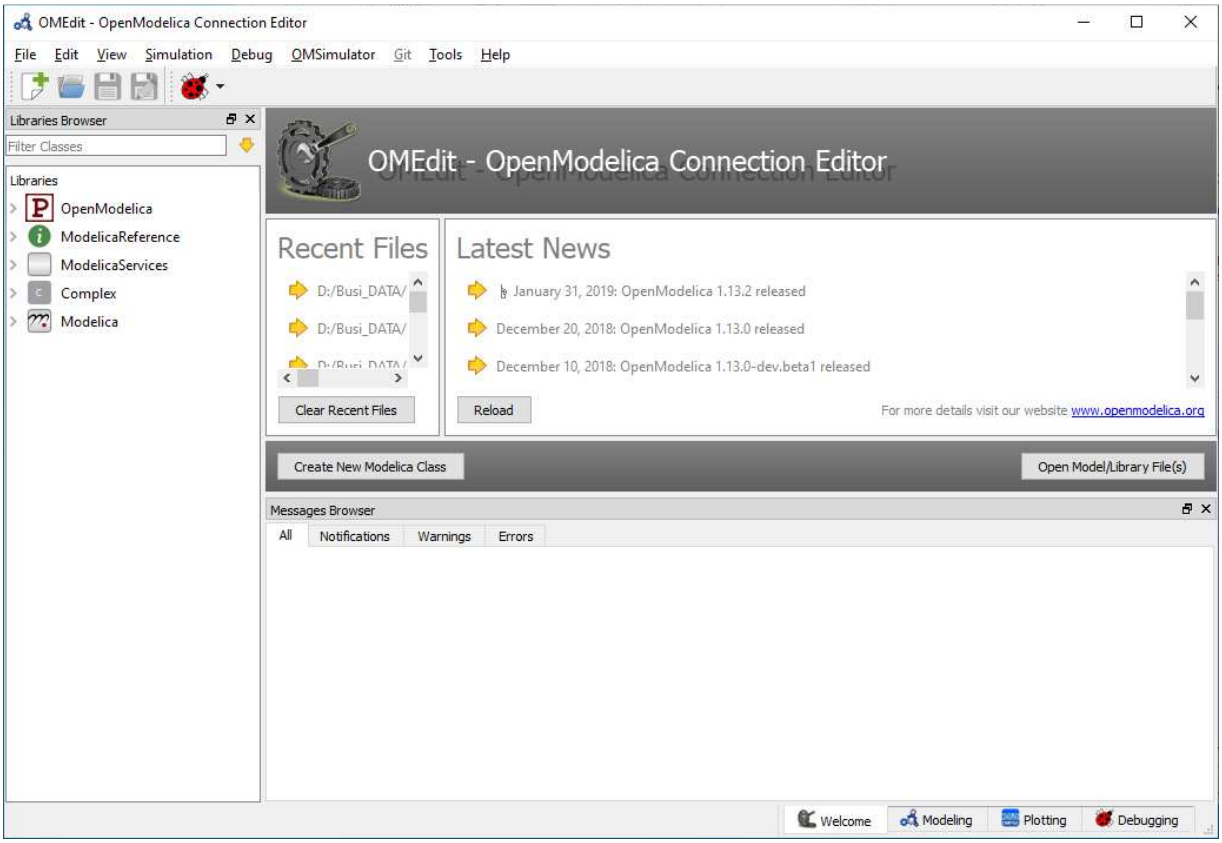


Figure 1: Default OMEdit View

This view is divided into the top tool sections, the libraries on the left and the main window that contains the 'Welcome' tab and the 'Messages Browser'.

Along the bottom are the tabs for the 'Welcome' Page (currently shown), the 'Modeling' page where models and similar items are modified as well as the 'Plotting' and 'Debugging' pages. The Debugging options are not discussed in this document beyond the basic usage when errors are found.

Generating a new system is done using the File > New Modelica Class option, the Ctrl+N shortcut or corresponding button below. Opening an existing model is accomplished using the same approach. Note, however, that for models that consist of multiple, custom components that may have been separately grouped into other classes, all relevant classes need to be loaded as will be discussed in the next section.

If no class has been loaded (whether opening an existing one or creating a new one) the Modeling tab will not show any options.

Opening an existing Class

By clicking the 'Open Model/Library File(s)' button and selecting a file, the selected items are shown below the standard Libraries as shown in Figure 2:

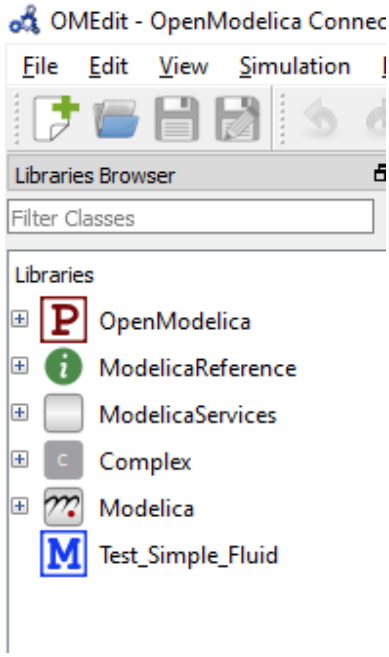


Figure 2:Test_Simple_Fluid Model loaded

Double clicking the Model file opens it up in the Modeling Tab. If the model contains items that are not part of the standard library (i.e. if it contains models that are made up in a customized fashion from existing/other models), then the display will show that the corresponding item is not available as is the case in Figure 3:

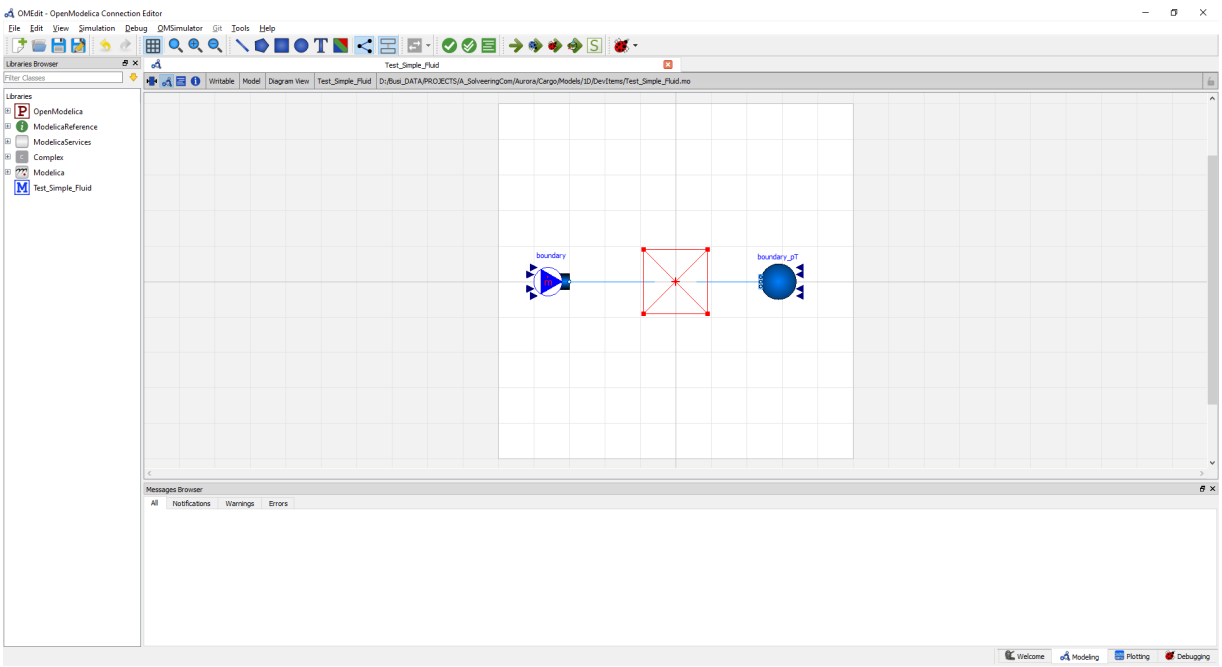


Figure 3: Test model with missing sub-model

Here, there is a missing item which gets represented by a red box with an X in its center. Usually, the missing component is simply a different class/set of files that wasn't loaded. By loading the missing class, the original component is updated, as shown in Figure 4:

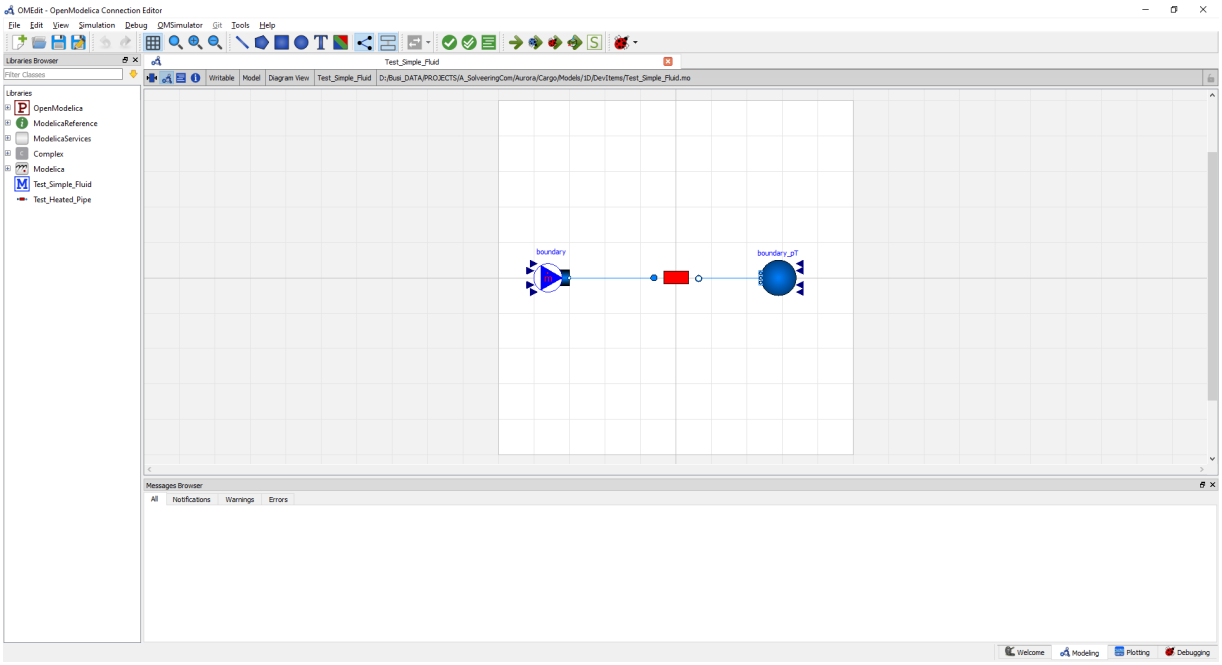


Figure 4: Model with all dependent classes loaded

In the next section, the steps to generate this very simple model are presented.

Creating a Fluid system

We start off by creating a New Model and call it "Test_Simple_Fluid". The type is left as 'Model' and for now no other options are set.

In the blank space we want to add a fluid source and sink. For the purpose of this example, the source will be a prescribed mass flow source and the sink will be a boundary with a prescribed pressure and temperature. These components are found in the 'Modelica' root library under the Fluid group: Modelica.Fluid.Sources.MassFlowSource_T and Modelica.Fluid.Sources.Boundary_pT (the dot notation is what the application internally uses to determine where in the library tree the classes are found, for the graphical representation, each dot represents a sub-class of is parent class).

Upon dragging and dropping the corresponding items from the library tree into the Modeling page, the application asks for a name to be assigned to the component. These names should be such that they are later easily identified as errors are more easily resolved by looking at the component that is causing the issue and determining whether perhaps there is an error in the units or similar.

Each component can also be dragged around, rotated (Ctrl+R) or flipped vertically or horizontally (V or H respectively). At this point, the modeling page should look similar to Figure 5:

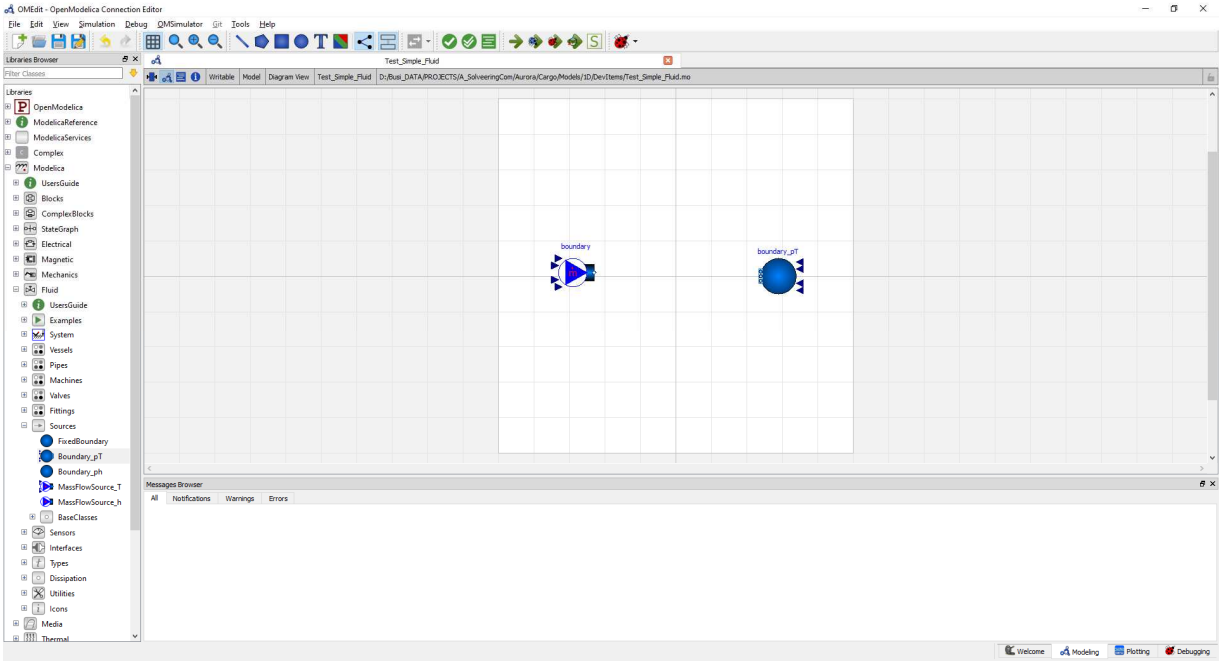


Figure 5: Example with source/sink components added.

Before adding a simple pipe with heat flowing into it, it makes sense to discuss encapsulation. Frequently there are cases where a more complex system is to be represented by a number of simpler sub-components or items that are not present in the library. For these items, it is advisable to generate a model that can be re-used in a way that allows flexibility and at the same time also better debugging of the overall model. For this purpose, separate library items may be created. Here, the simplest combination of a pipe (from the library) together with a fixed heat source (also from the library) will be combined into a Heated Pipe model. To do this, a new model is opened and a pipe (Modelica.Fluid.Pipes.DynamicPipe) and a Fixed Heat source (Modelica.Thermal.HeatTransfer.Sources.FixedHeatFlow) are added. In addition, since this is not a fully defined model on its own, two fluid interfaces are added, one shown as an upstream and one as a downstream port (though it does not matter which is upstream and which is downstream unless the direction is fixed). These components are found in the Modelica.Fluid.Interfaces.FluidPorts_a and .._b classes. The setup is shown in Figure 6:

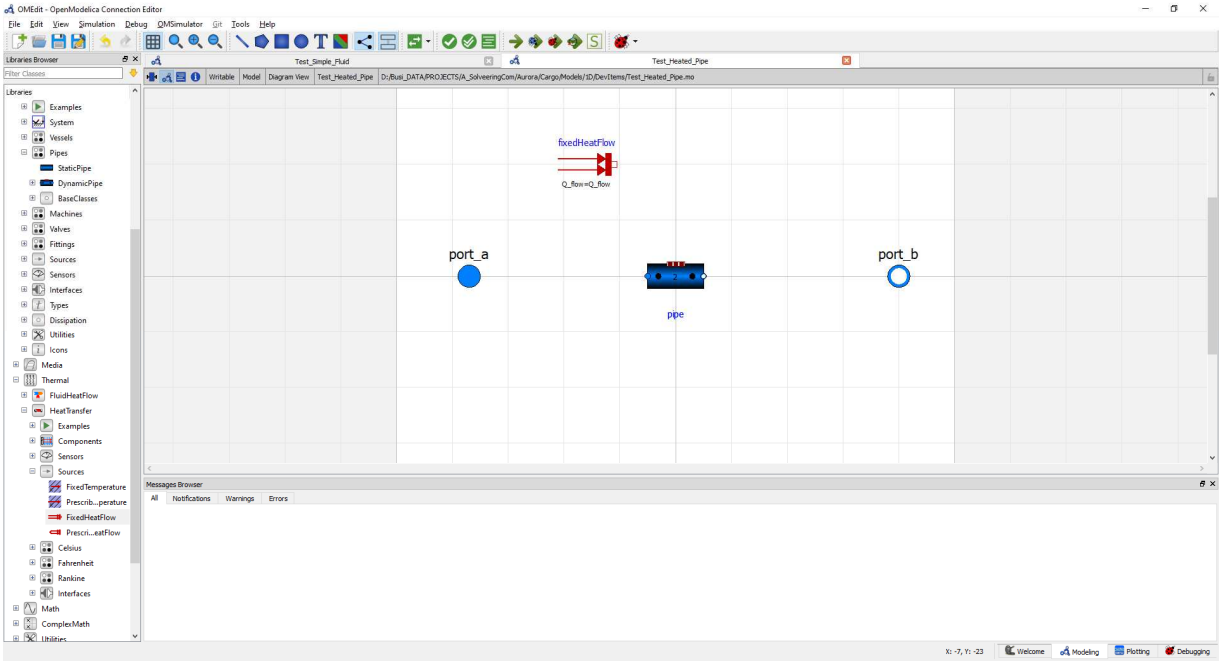
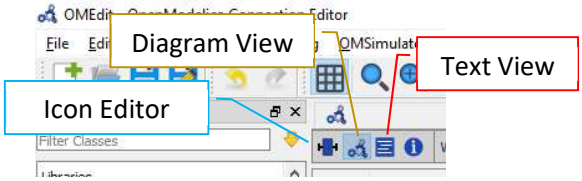


Figure 6: Component placement for the Heated Pipe sub model

The next step is to connect the items to determine which items are connected to which. For this, hover the mouse over the ports (red squares for HeatTransfer items, blue circles for Fluid items). The mouse cursor should change to a 'cross-hair' shape. By clicking-dragging (holding the mouse down initially), a connector is started. Clicking anywhere other than on another port creates a corner in the connection. Clicking on another port completes the connection. If the target of a connection is a multi-port (which is the case for the HeatTransfer port on the pipe), the application asks for the Index of the connection. If this is the only connection, then the index is "1". If there are multiple connections, the index will generally be the next higher. Note that currently it appears that there is a bug in that if the connection goes from the 'port_a' to the pipe, the connection is black, otherwise blue (blue indicating a fluid connection whereas black is generally for scalar values).

Once the three connections are made, the sub-model is essentially done. However, attempting to run this model would generate problematic results as the model does not specify the type of fluid being considered nor any relevant parameters of the pipe nor the amount of heat flow. It would be simple to modify the values directly by right clicking on the pipe and heat source and updating the 'Parameters' values. In order to make this model more flexible, the parameters will be exposed to any parent classes that implement it. Further, the fluid will equally be set as a default and may be modified by parent classes.

In order to edit the code of the class, we switch from the Diagram view to the Text View:



The Text (Code) view for this simple model is shown in Fig:

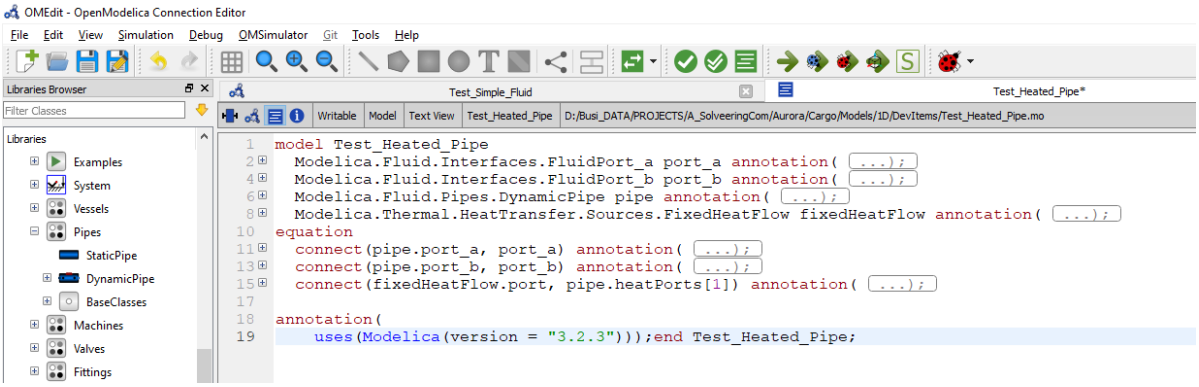


Figure 7: Code view for heated Pipe

The code uses the Modelica language and the Diagram view and the Text view are directly dependent upon each other (any changes here are reflected in the diagram if they are of a graphical nature).

The items we want to introduce are the fluid definition, heat flow and pipe diameter/length (roughness, etc. are not modified externally for this example). In order to add parameters, the following items are added between the 'fixedHeatFlow' component and the 'equation' section:

```
parameter Modelica.SIunits.HeatFlowRate htFlow =100 "Heat Flow into the fluid";
parameter Modelica.SIunits.Length pipeDia =0.0254 "Diameter of pipe";
parameter Modelica.SIunits.Length pipeLen = 10 "Length of Pipe";
```

Here the keyword 'parameter' is used to define an item (a parameter) that is not connected like the other components and, as it is located in the top section, available to external/parent models. The Modelica.SIunits.Length is the type followed by the name given to it. Providing a default value is optional but generally recommended. Adding a string section with a comment that explains the parameter is equally recommended unless the name makes this obvious. If the parameter is to be of a scalar value, the type is "Real", "Integer", "Boolean" or "String", depending on use. If the units are to be considered (and in some cases they can also be externally chosen), then it generally makes sense to use the built-in SIunits. Note that each line must be terminated by a semicolon.

If there are a lot of SI units, then the term can be abbreviated by inserting the line

```
import SI = Modelica.SIunits;
```

at the top of the model. By doing so, the first parameter can be shortened to

```
parameter SI.HeatFlowRate htFlow =100 "Heat Flow into the fluid";
```

Alternatively, it is also possible to just use “import Modelica.SIunits.*;” and therefore omit the use of the “SI.” part.

The inclusion of the fluid definition is accomplished in a somewhat awkward way. Adding a line

```
   replaceable package Medium = Modelica.Media.Air.DryAirNasa constrainedby
Modelica.Media.Interfaces.PartialMedium;
```

adds an item called Medium that can be included anywhere that the class uses a Medium parameter. For example, the Dynamic Pipe is told to use (replace) its internal fluid definition by:

```
  Modelica.Fluid.Pipes.DynamicPipe pipe(redeclare package Medium = Medium)
```

Here the pipe already has an internal definition of a medium (which it uses to look up relevant properties) but is now told to replace it with whatever is being provided, in this case Air, defined by the DryAirNasa functions for the properties. Since the Medium package itself is ‘replaceable’ the parent class can override this too, therefore allowing a hierarchical structure to declare fluid properties

Similarly, the other parameters are also included in both the pipe and the heat source:

```
   Modelica.Fluid.Pipes.DynamicPipe pipe(redeclare package Medium = Medium, diameter =
pipeDia, length = pipeLen , use_HeatTransfer = true);
  Modelica.Thermal.HeatTransfer.Sources.FixedHeatFlow fixedHeatFlow(Q_flow = htFlow);
```

Essentially, the default components (pipe, fixedHeatFlow) are modified with new parameters (given in parentheses). For sake of clarity, the annotation parts have been omitted.
Note that the pipe also has the ‘use_HeatTransfer=true’ parameter added which can be done either from the code window or by clicking on the component and then changing the setting from the ‘Assumptions’ page.

In order to define defaults, such as gravity, start states and similar, a standard object of type Modelica.Fluid.System needs to be dragged into the model. Note that this item generally defines initial conditions and whether flow can be reversed and should thus, for a sub-model like this, be modifiable from a parent if necessary. For this purpose, the line

```
  inner Modelica.Fluid.System system annotation(
    Placement(visible = true, transformation(origin = {-42, -30}, extent = {{-10, -
10}, {10, 10}}, rotation = 0)));
```

needs to be modified to not just use ‘inner’ but ‘inner outer’ which allows the system component to be modified as necessary.
The full text (code) is given here:

```
model Test_Heated_Pipe
  import SI = Modelica.SIunits;

  replaceable package Medium = Modelica.Media.Air.DryAirNasa constrainedby
Modelica.Media.Interfaces.PartialMedium;

  Modelica.Fluid.Interfaces.FluidPort_a port_a annotation(
    Placement(visible = true, transformation(origin = {-74, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0),
iconTransformation(origin = {-74, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  Modelica.Fluid.Interfaces.FluidPort_b port_b annotation(
```

```
    Placement(visible = true, transformation(origin = {80, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0),
iconTransformation(origin = {80, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0)));

  Modelica.Fluid.Pipes.DynamicPipe pipe(redeclare package Medium = Medium, diameter = pipeDia, length = pipeLen
, use_HeatTransfer = true) annotation(
    Placement(visible = true, transformation(origin = {0, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  Modelica.Thermal.HeatTransfer.Sources.FixedHeatFlow fixedHeatFlow(Q_flow = htFlow)  annotation(
    Placement(visible = true, transformation(origin = {-32, 40}, extent = {{-10, -10}, {10, 10}}, rotation =
0)));

  parameter Modelica.SIunits.HeatFlowRate htFlow =100 "Heat Flow into the fluid";
  parameter Modelica.SIunits.Length pipeDia =0.0254 "Diameter of pipe";
  parameter Modelica.SIunits.Length pipeLen = 10 "Length of Pipe";
  inner outer Modelica.Fluid.System system annotation(
    Placement(visible = true, transformation(origin = {-42, -30}, extent = {{-10, -10}, {10, 10}}, rotation =
0)));

equation

  connect(pipe.port_a, port_a) annotation(
    Line(points = {{-10, 0}, {-72, 0}, {-72, 0}, {-74, 0}}, color = {0, 127, 255}));
  connect(pipe.port_b, port_b) annotation(
    Line(points = {{10, 0}, {80, 0}, {80, 0}, {80, 0}}, color = {0, 127, 255}));
  connect(fixedHeatFlow.port, pipe.heatPorts[1]) annotation(
    Line(points = {{-22, 40}, {0, 40}, {0, 4}, {0, 4}}, color = {191, 0, 0}));

annotation(
  uses(Modelica(version = "3.2.3")));
end Test_Heated_Pipe;
```

The last item to complete is to insert this custom model into the parent class and to connect it:

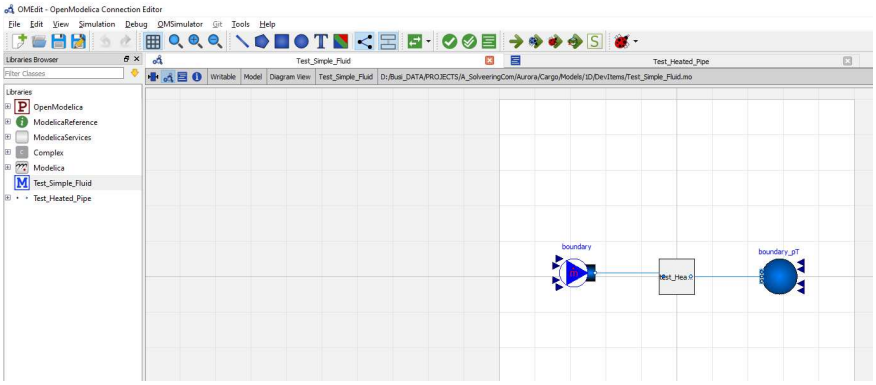


Figure 8: Test model with custom component

By double clicking the heated pipe model, the three parameters are made available. Lastly, the parent model needs to be updated by including the fluid definition in all components that are fluid-related. The full code is shown here:

```
model Test_Simple_Fluid
  replaceable package Medium = Modelica.Media.Air.DryAirNasa constrainedby
Modelica.Media.Interfaces.PartialMedium;
  Modelica.Fluid.Sources.MassFlowSource_T boundary(redeclare package Medium = Medium,T = 283.15,
m_flow = 0.1, nPorts = 1)  annotation(
    Placement(visible = true, transformation(origin = {-56, 2}, extent = {{-10, -10}, {10, 10}},
rotation = 0)));
  Modelica.Fluid.Sources.Boundary_pT boundary_pT(redeclare package Medium = Medium,T = 283.15,
nPorts = 1, p = 101325)  annotation(
    Placement(visible = true, transformation(origin = {58, 0}, extent = {{10, -10}, {-10, 10}},
rotation = 0)));
  Test_Heated_Pipe test_Heated_Pipe(redeclare package Medium = Medium) annotation(
    Placement(visible = true, transformation(origin = {0, 0}, extent = {{-10, -10}, {10, 10}},
rotation = 0)));
  inner Modelica.Fluid.System system annotation(
    Placement(visible = true, transformation(origin = {-42, -54}, extent = {{-10, -10}, {10,
10}}, rotation = 0)));
```

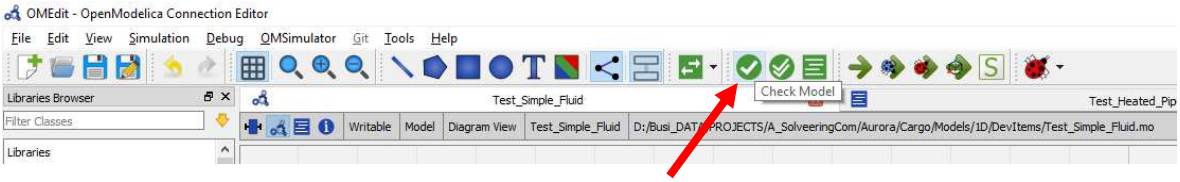
```
equation
  connect(boundary.ports[1], test_Heated_Pipe.port_a) annotation(
    Line(points = {{-46, 2}, {-8, 2}, {-8, 0}, {-8, 0}}, color = {0, 127, 255}));
  connect(test_Heated_Pipe.port_b, boundary_pT.ports[1]) annotation(
    Line(points = {{8, 0}, {46, 0}, {46, 0}, {48, 0}}, color = {0, 127, 255}));

annotation(
  uses(Modelica(version = "3.2.3")));

end Test_Simple_Fluid;
```

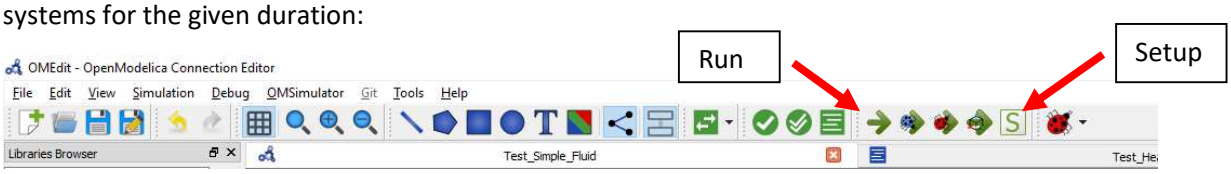
Running

Before running a model, it usually makes sense to check that there are no code errors or issues with the setup. This is done using the 'Check Model' option at the top:



This will check the model and related sub-models for any issues in definition. It will provide feedback by showing the number of equations and variables, which should be the same (otherwise it cannot be solved).

Pressing the Run button will compile the code into an executable and will then attempt to solve the systems for the given duration:



The setup will allow different parameters to be modified for the run, especially the start and stop time and the number of steps in between (though the solver may run more steps in order to obtain a solution where data changes trends). Modifying the Setup page and clicking OK will generally run the model unless the checkbox at the bottom of the page is unchecked.

During the run, the Messages Browser will show notifications and errors if there are any. If there are no compilation/setup errors, an additional output window will open up (this may take some time depending on the size of the model) and information about the progress is shown. Upon completion, the Plotting window with the results of the model simulation (the output window will show "The simulation finished successfully" on its last line).

Debugging

In some cases, usually for anything that isn't trivial, error messages will prevent successful completion of the simulation run. In these cases, debugging the model becomes necessary.

Though there are built in tools that provide some support, debugging is not approached in the same way as for software development as there is no step-by-step execution possible (other than for algorithms, which is not discussed here). Instead, the best approach is generally to look at the error messages

provided and, if there is sufficient information to trace the issue to a specific component, look at whether the defaults or provided parameters are reasonable and make sense. Otherwise, an approach is recommended that gradually builds the model, attempting to start with the simplest possible layout and then gradually building it into a more complex system from there, all the time testing the model to check for errors and to get an understanding of what parameters affect what (negative values can be an issue, depending on where they appear).

Beyond this, there is some built in debugging capability that can be called from the output window for errors where a 'debug more' option is shown (and provided as a hyperlink to further information).

Post Processing

Once a run completes successfully or at least partially, the display switches to the 'Plotting' window and the most recent results are loaded (if multiple models have been run, their results may still be displayed). The results for this case are shown in Figure 9:

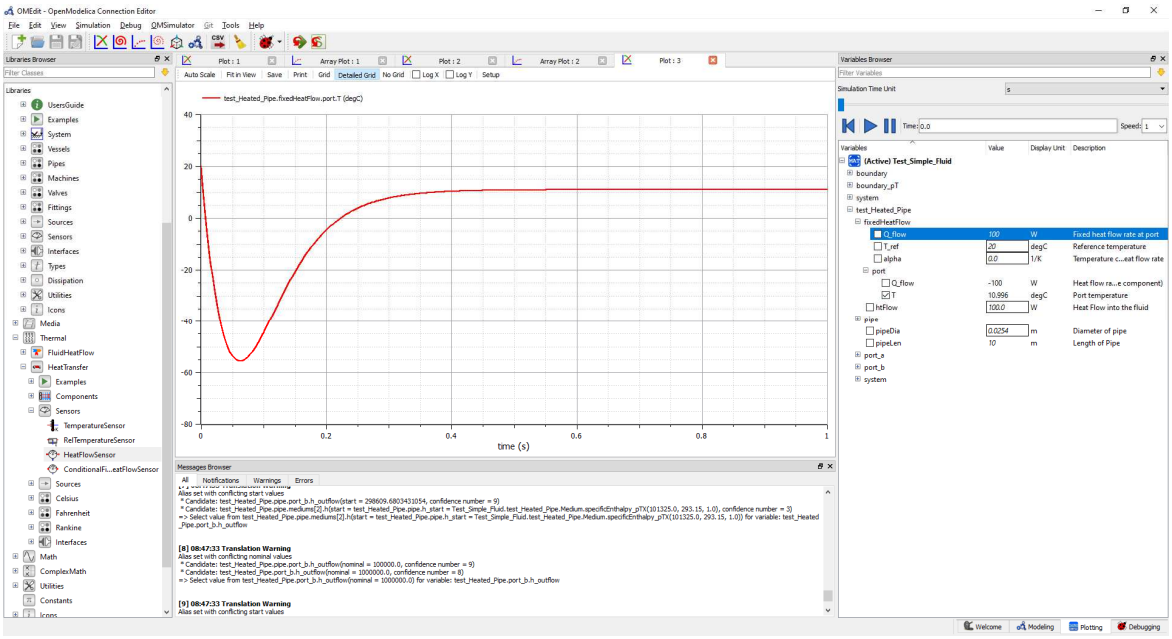


Figure 9: Restuls in Plotting Window for test case

In the right-side panel, the available parameters are shown in a hierarchical structure. For this case, the temperature T of the heated Pipe sub-model are plotted which are associated with the heat port of the fixedHeatFlow component. Note that the same temperature may be shown for the pipe, however, the pipe has two slightly different temperatures associated with it due to its underlying model.

In several situations, the amount of data available in this manner is overwhelming and often unnecessary. For this purpose, the developer of the class may choose to place certain components within a 'protected' section (preceding the 'equation' section in the code). Items placed between these items are not visible outside of the class that uses it and can thus clean up the output generated

Working with Models, Sub-Models and Sensors

With any sufficiently large model, organization becomes an important aspect which makes both working with the model (updating, modifying, running) as well as debugging issues easier. The first step is to determine a suitable architecture for the model in question. In some cases, it is best to keep the entire system "flat" with all the items contained in a single model. In other cases, it is best to build a hierarchy whereby several components may be re-used or simply encapsulated so as to keep the structure clean. The decision which to use (or to what extent) is something that generally only comes with experience though generally breaking the models up into a large number of sub-models also makes debugging easier as issues can be attacked using a divide-and-conquer approach (each sub-model can be set up in a separate test case and simulated without the complexities of the larger model in question. If the sub-model works and can perhaps be verified, then there is a good chance that it works as intended, provided that the test case is sufficiently broad to capture possible corner cases).

Once a model is compiled to a larger system/model, the simulation may take a long time, so running smaller cases may actually accelerate the process.

Once the larger model has been completed, working on the sub-models (this also applies to existing/default libraries, not just custom items) can be easily achieved by right-clicking on the component in question and selecting 'Open Class' to access the details of that model (and also allow changes to how the model is represented by modifying its 'Diagram View' item or similar). If the top-level model uses multiple sub-models of the same type, then making a change to one automatically updates all of them, making a strong case for breaking up models in a logical way.

Beyond the model/sub-model approach, the question of what parameters are of interest is something that should be answered at an early stage. Looking at the example developed in this document and the results obtained from it, it becomes clear that a number of items may not have been of interest. Though having all parameter/variable values available is useful for debugging, in some cases this can be overwhelming. In order to reduce the amount of data, pertinent items should be considered and others perhaps hidden away (defined in the 'protected' section if applicable). For the case of the heated pipe, if the only items that were of interest for this custom pipe are the temperature at the port and the flow rate then sensors can be added to the model that specifically expose those variables (even in different units for the temperature sensor). Now it is possible to hide the other items of the pipe and only show the sensors as public items. For some components, this will be tricky as any items that need to be externally connected (the Modelica.Fluid.Interfaces.FluidPorts in this example) cannot be defined in the Protected section since they need to be generally available. The temperature of the pipe (and the pipe itself) as well as the heat flux however can easily be defined here since they only need to exist in the sub-model and are not directly accessed via the parent model. By applying this methodology to sub-models as well as their sub-models, the overall set of results can be cleaned up to only those items that are desired.

For such an approach, the use of the Protected section is ideal, especially as it can easily be commented out (using two forward slashes "//") during initial debugging and then made active again once the model works satisfactorily.

In terms of use of existing library components, a general discussion is perhaps in order. One item to keep in mind as these components are used is that certain components need to be interfaced/connected to components of a specific type that matches the port to which they are connected (otherwise the application will provide an error during compilation). It does not make sense to connect a heat source to the fluid side of a pipe, just like connection of a fluid source to the heat port is inconsistent. For most of these types, the matching item is fairly clear. In other cases, however, the purpose of the connection needs to be considered. In the example, the flow rate for the mass flow source was simply entered as a constant value in the component's parameter page. This is fine for test cases but the value may frequently be provided by other items. Here the external connection is used (and with this specific component, the Boolean value is set to use the external parameter value rather than the internal value, something that can be a source of errors) and a value provided of type 'Real'. The value can come from another component or itself be provided by a constant. If it is a constant that is defined as a block, then the Modelica.Blocks items are to be used, specifically those of type 'Sources'. Similarly, a number of types contain sources that are to be considered for the purpose of providing flow, values or similar based on a definition.

References

[1] "Dr Modelica," [Online]. Available: http://omwebbook.openmodelica.org/DrModelica.

[2] "OpenModelica Reference," [Online]. Available: https://build.openmodelica.org/Documentation/ModelicaReference.html.

[3] "Modelica Language Specification," [Online]. Available: https://www.modelica.org/documents.

[4] "Maplesoft documentation," [Online]. Available: https://www.maplesoft.com/documentation_center/online_manuals/modelica/Modelica.html#Modelica.

[5] "OpenModelica," [Online]. Available: https://www.openmodelica.org/.