TROUBLESHOOTING

Lecture 2.4

Modelon

OVERVIEW

- Development - Best practice
- Troubleshooting
- Diagnostics

DEVELOPMENT

BEST PRACTICE

SPECIFY

- When creating a new component or system, start by defining (at least) one test case that specifies how the model should function:
  - It should excite your model in a way you easily can verify
    - For a spring, plot the force compression curve
  - For some configurations, the result can be compared to an equivalent test using another model
    - If it is a nonlinear spring, parameterize as linear and compare with a standard linear spring.
  - It excites the whole operating region of the model
    - Make sure that the spring is tested for both rebound and compression
  - It verifies the behavior under different causality conditions
    - For a nonlinear gear, make sure that the results and the generated code is consistent, independent of which flange is dependent on the other.

IMPLEMENT

- Take advantage of the benefits of component oriented modeling, separate code into reusable primitive models
  - Use a base class that defines the relations between connectors, so that only the force-compression relation has to be given for each new spring model
- Use types, units, enumerations and a stringent nomenclature
  - “if spring_type == Types.Spring.NONLINEAR then ...” is easier to maintain and read than “if spring_type == 1 then ...”
  - For a spring compatible with Modelica.Mechanics.Translational, use s for position, v for velocity, a for acceleration and f for force.
- Define a model, either with components and connections, or with equations/algorithms.
  - A mix between the concepts makes it much harder to understand and debug

MAINTENANCE

- Version your models
  - Use a version management system, e.g. SVN, Git to be able to back-track changes
- Collect test cases and reference trajectories systematically
  - Once a model is finalized and the test results are fine, store the cases and the results for future reference
- Rerun the tests to detect unexpected changes/errors as early as possible
  - A good practice is to rerun test either on a regular basis or when a change has been made to the code
- There are tools to support automation of the maintenance
  - Modelon's Model Testing Toolkit - MTT (cross platform regression testing)
  - Integrate with Jenkins to automate the testing procedure

TROUBLESHOOTING

TRANSLATION PROBLEMS

- Case: Cannot translate the model.

- Common issues:
  - Missing class or parameter declaration
  - Modifiers on missing parameters or components
  - Missing connection
  - Number of equations and variables do not match
  - Modelica syntax problems
  - Index reduction problems
  - Overspecified number of initial values
  - Overspecified number of states

FINDING THE PROBLEM

- Read the Translation log
  - Be aware of any Warnings given in the Translation log
  - Try and see if you can locate the part of the model that does not work
- Check your model
  - Also try and check subparts of your model

- When you isolate the part of the model that does not translate, it might be a good idea to create a smaller test model for that part.

- If the error is related to Modelica syntax, refer to:
  - Modelica Language Specification

SIMULATION PROBLEMS

- Case: The model translates but does not run properly.
- Common issues:
  - Initialization problems
    - Convergence problems
    - Solver finds the wrong initialization point
  - Simulation does not finish due to:
    - Convergence problems in nonlinear systems
    - Singularities
    - Too stiff, stepsize goes to zero
    - Division by zero
  - Unphysical results
    - Missing parameter propagation
    - Using wrong units
  - Performance
    - Simulation speed too slow

DIAGNOSTICS

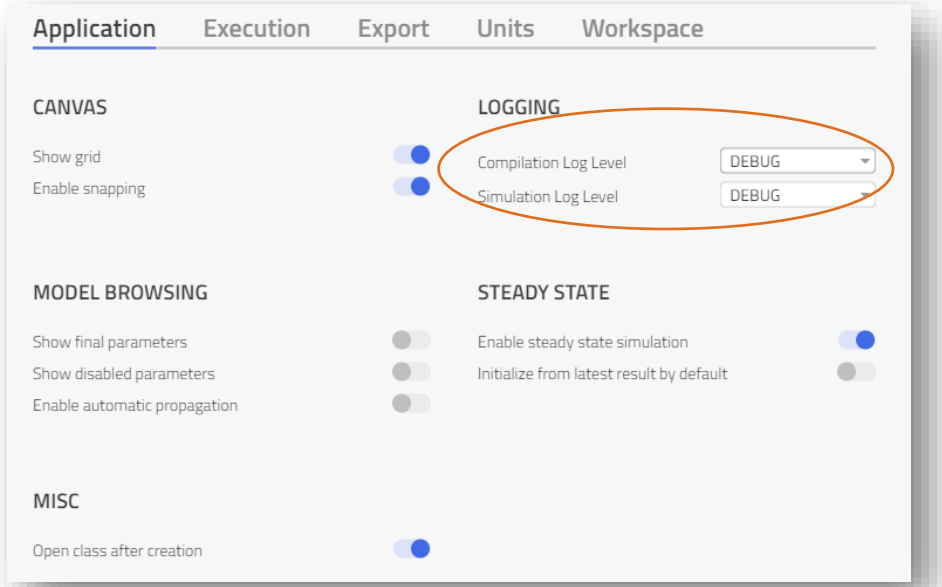
SETTINGS THAT CONTROL DIAGNOSTICS

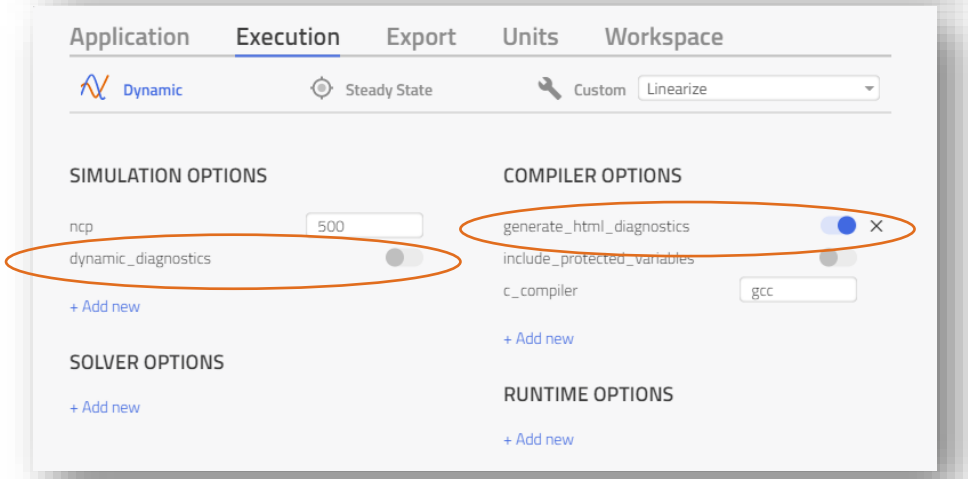
Application tab

- "Compiler Log Level" and "Simulation Log Level"
  - Controls amount of information written to logs

Execution tab

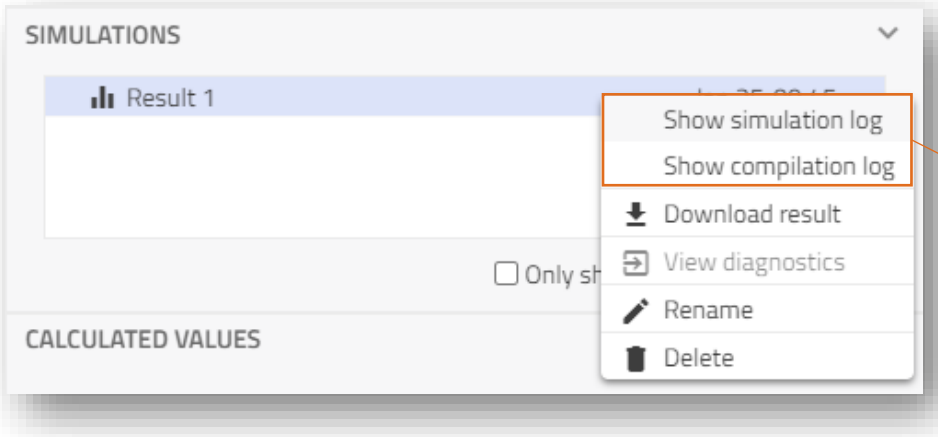
- "`generate_html_dignostics`"
  - Generates browsable pages containing information about the compilation
- "`dynamic_diagnostics`"
  - Adds additional diagnostic variables to the result

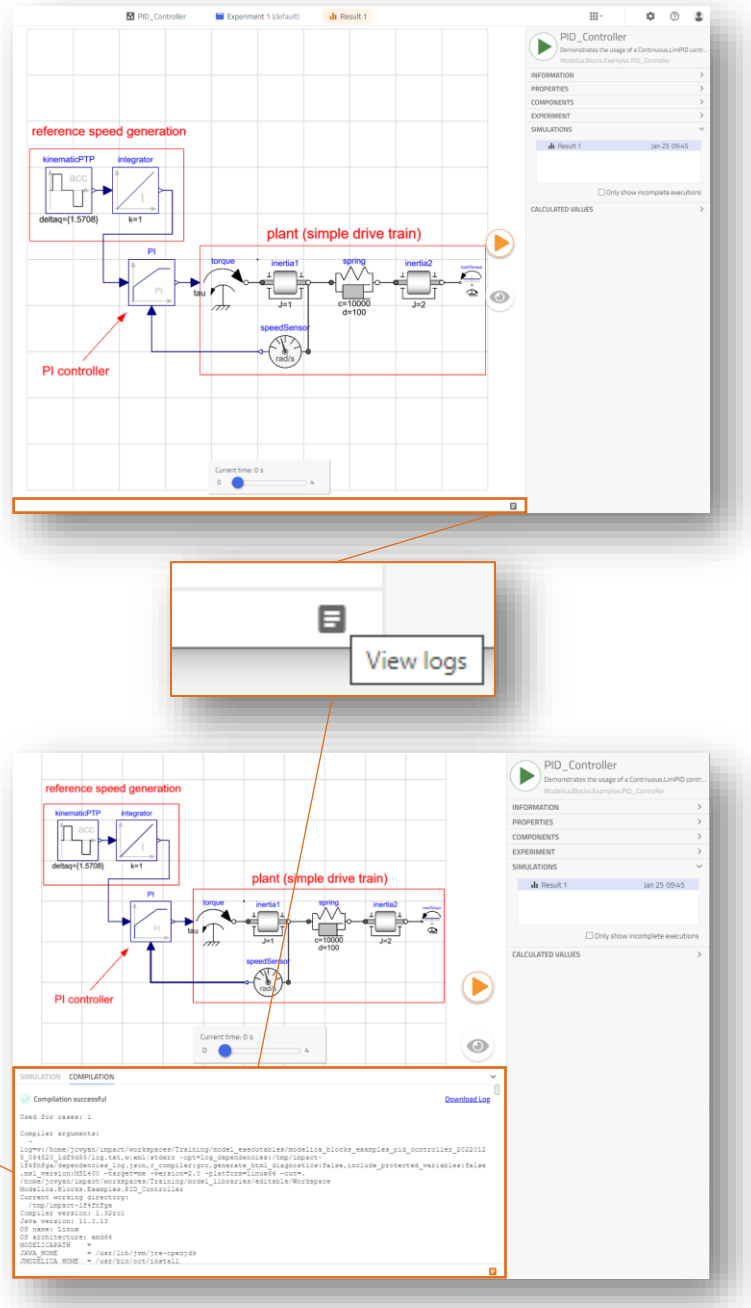




ACCESSING LOGS

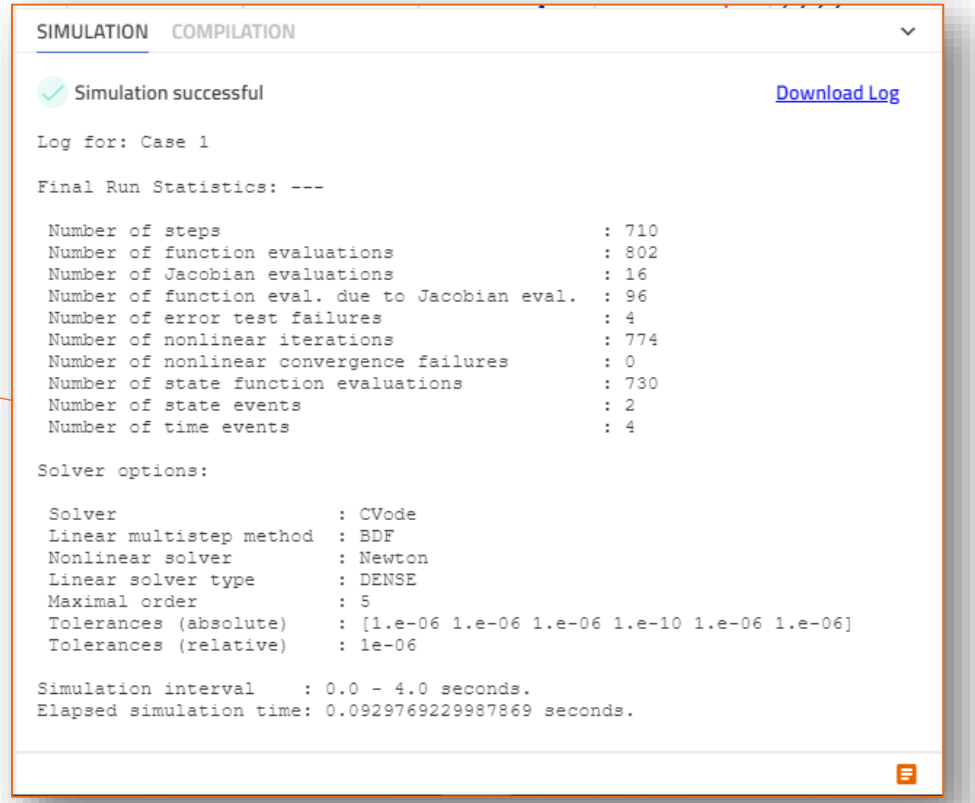
- After Compilation and Simulation have finished, compilation and simulation logs can be accessed by right clicking the result entry
- They also can be accessed from the logs bar at the bottom of the canvas

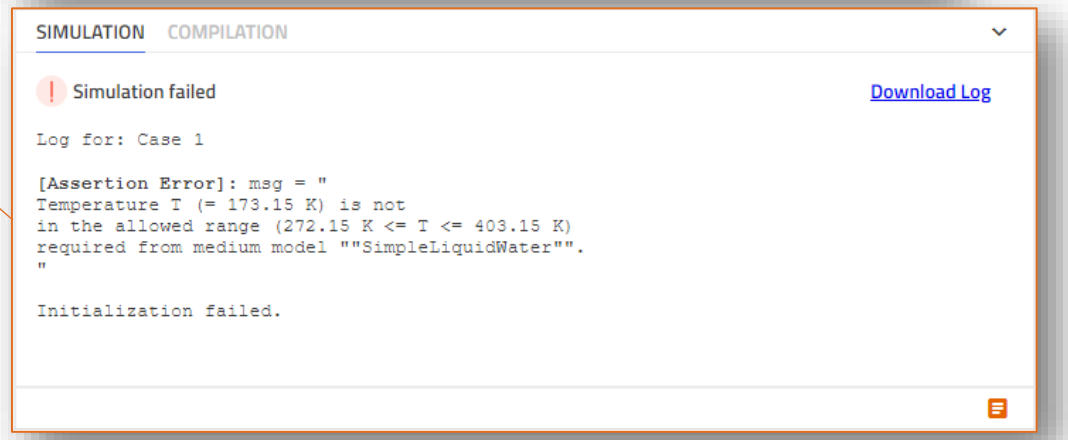




SIMULATION LOG

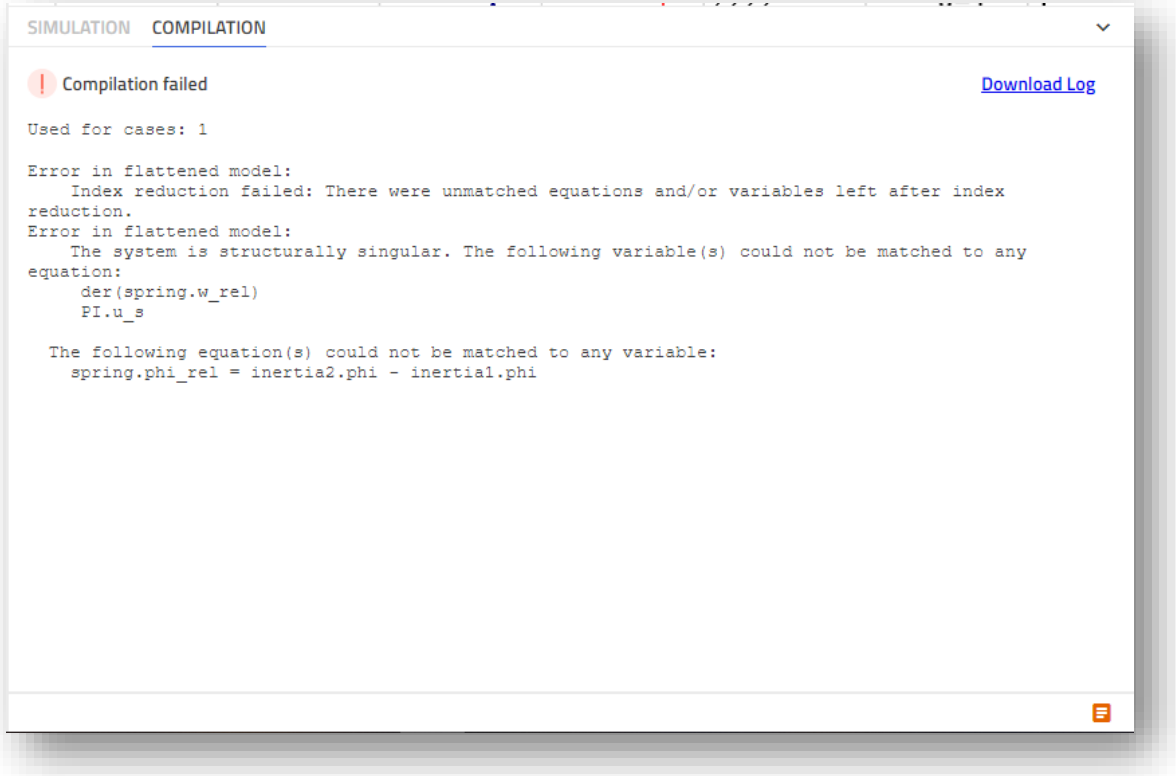
- Contains selected solver options and run statistics from the simulation if simulation run successfully.
- In case of simulation failure, it will give info about the error occurred.
- Errors found here includes:
  - Initialization failures
  - Convergence failures
  - Division by zero
  - Assertion errors





COMPILATION LOG

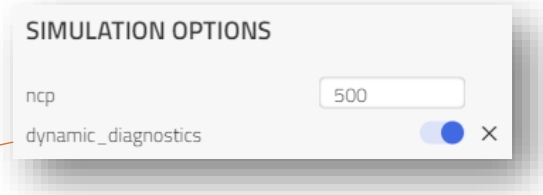
- Contains output from the compilation process according to log level set.
- Errors found here includes:
  - Syntax errors
  - Structural errors
    - Missing connections
    - Number of equations and variables do not match
  - Index reduction problems
  - Over specified number of initial values

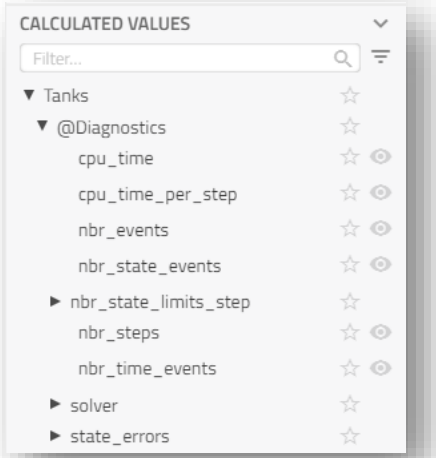


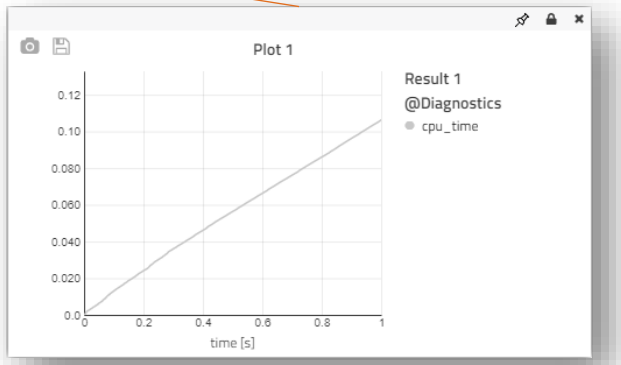
DYNAMIC DIAGNOSTICS

If "`dynamic_diagnostics`" is activated:

- Extra diagnostic variables are added to the result:
  - **cpu_time**: elapsed cpu time
  - **nbr_events**: number of events triggered
  - **nbr_state_limits_step**: number of times the error of a step limited the step size of the solver.
  - **state_errors**: error estimates of all state variables
- Like any other variable, drag the diagnostic variable to canvas to plot or add a sticky.

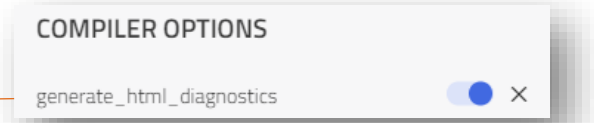




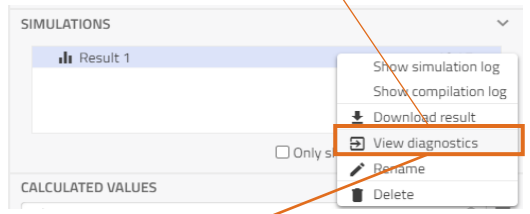


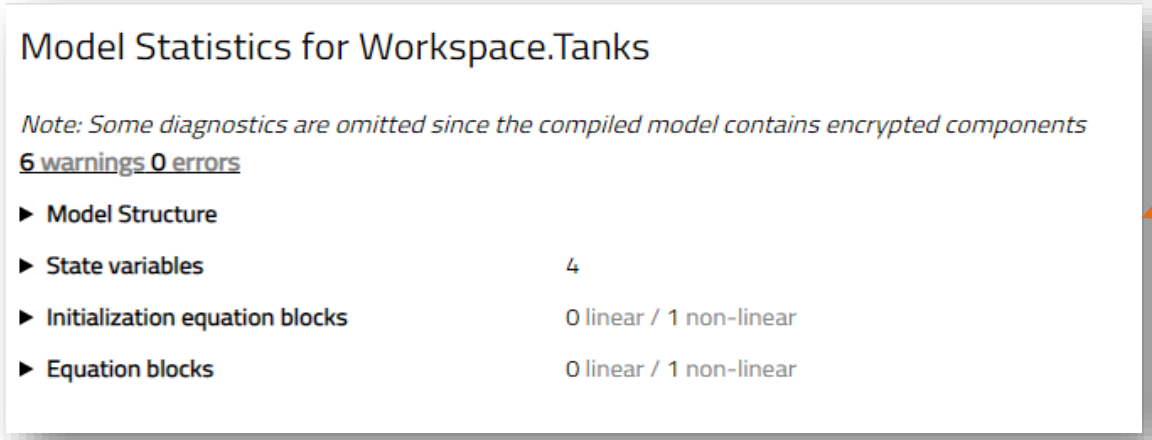
ADVANCED MODEL DIAGNOSTICS

If "`generate_html_dignostics`" is activated:



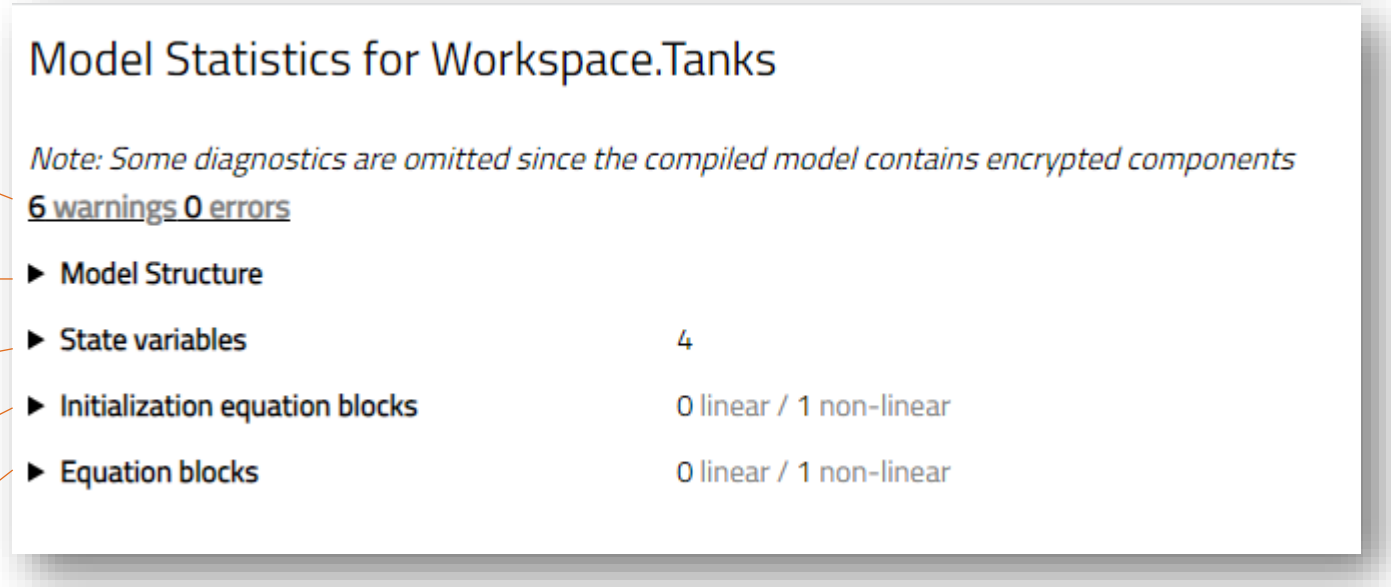
- Additional compilation diagnostics and information is available from "View diagnostics":





ADVANCED MODEL DIAGNOSTICS: OVERVIEW

- Lists all compiler errors and warnings that occurred during compilation (same as compilation log).
- Information about the model structure such as number of equations, variables etc.
- List of selected state variables
- Information about initialization equation system
- Information about equation systems in model



ADVANCED MODEL DIAGNOSTICS: MODEL STRUCTURE

Model Structure:

- Summary statistics of the compiled model after transformation

Full statistics:

- Full model statistics from before and after transformation.

  Model before transformation:

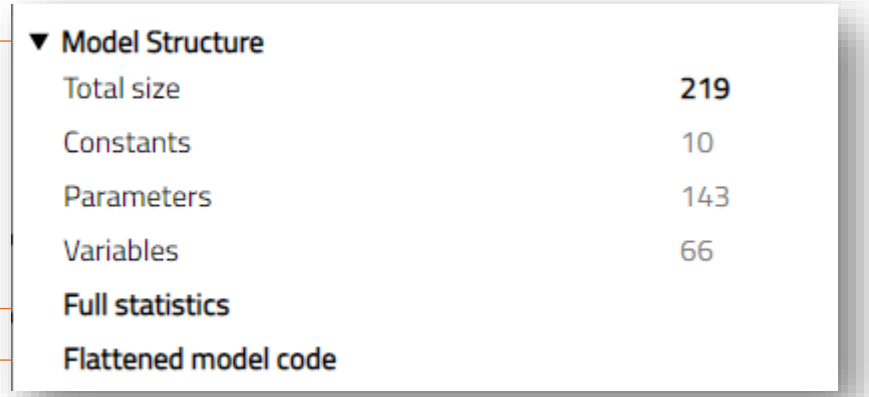
  - Summarizes model statistics of the flattened model.

  Model after transformation:

  - Summarizes the statistics of the model after the compiler has completed the transformation work (alias elimination etc.)

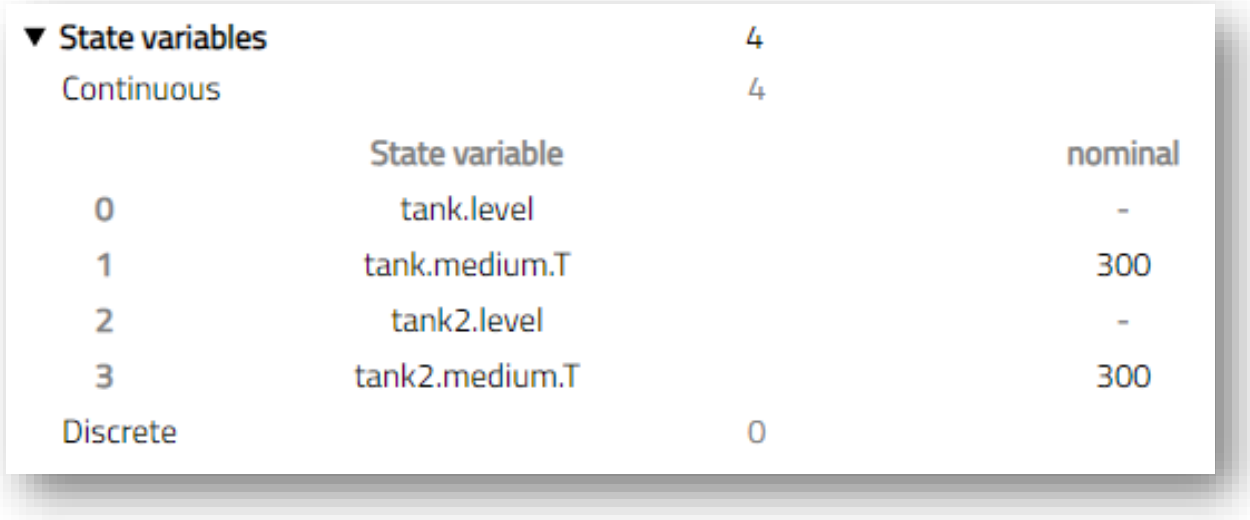
Flattened model code:

- Displayed flattened model code. The flattening process includes:
  - Expanding inherited classes (extend statements)
  - Replacing connect statements with actual equations
  - Etc.



ADVANCED MODEL DIAGNOSTICS: STATE VARIABLES

- Lists selected state variables
- Check so that the states selected are the expected ones. See lecture 2.2 for some advice about state selection.



▼ State variables 4

Continuous 4

	State variable	nominal
0	tank.level	-
1	tank.medium.T	300
2	tank2.level	-
3	tank2.medium.T	300

Discrete 0

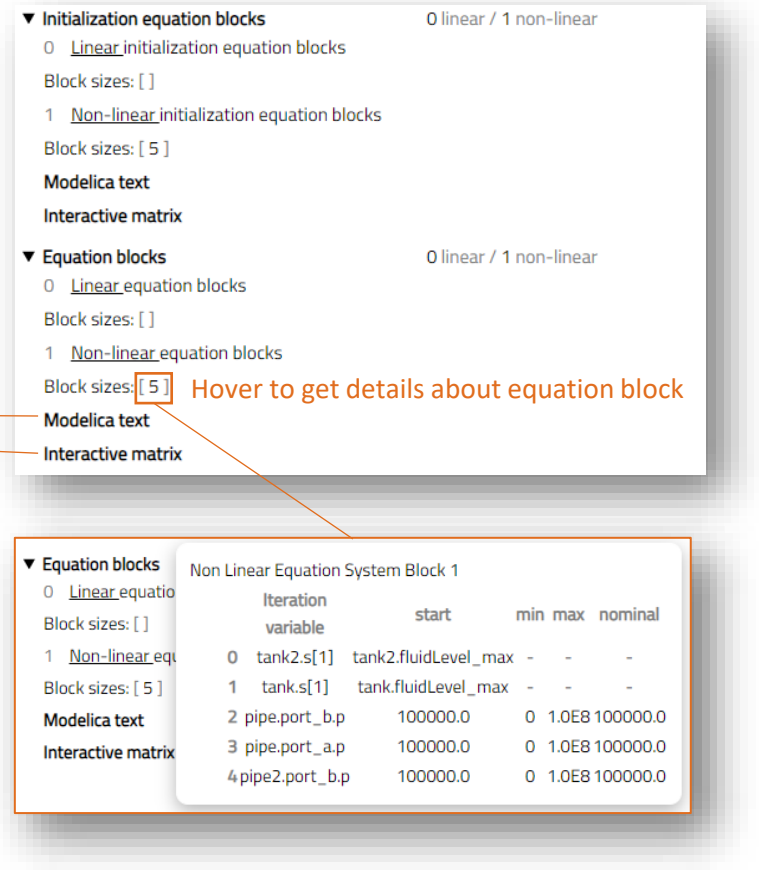
ADVANCED MODEL DIAGNOSTICS: EQUATION BLOCKS

(Initialization) Equation blocks:

- Displays detailed information about existing equation blocks in the model.
  - Non-linear equation blocks require iterative algorithms to be solved which effect both simulation speed and robustness (might not be able to find solution)
    - If possible, these should be kept to a minimum

Modelica text & Interactive matrix

- All equations and equation blocks are listed and sorted in the order in which they are calculated, i.e., by
  - Can be viewed both in text form (**Modelica text**) and block-lower triangular form (**Interactive matrix**)



WORKSHOP 2.4

In this workshop you will:

- Look at a simple steady state example
- Inspect the model using “HTML diagnostics”