

# MESSINA Tutorial



**Inhaltsverzeichnis**

Inhaltsverzeichnis	2
Tutorial	4
Example: Temperature Compensation	4
Test Case Description	4
Step 1 - Prepare the Test	6
Step 2 - Create New Project	9
Step 3 - Add Model and Configuration	12
Add a System Configuration to the Testenvironments	12
Add a Matlab/Simulink Model to the Configuration	14
Step 4 - Adding Signals to the Signalpool	17
Add Signals to the Signalpool	17
Step 5 - Add, Configure and Start a Target	19
Add and Configure a Target	19
Step 6 - Add a Visualization and Test Model	21
Add a Table Viewer Visualization	21
Testing the TempComp Model	22
Step 7 - Add a Control Panel Visualization	24
Add a Control Panel Visualization	24
Connect Signals to the Elements	27
Testing the TempComp Model	29
Step 8 - Create and Program a Test Case	30
Add a Test Case	30
About the preCondition and postCondition	34
What do we want to do?	34
Programming the preCondition Function	34
Programming the run Function	35
Programming the postCondition Function	35
Step 9 - Execute the Test Case	39
Execution of a Test Case - Table View	39
Execution of a Test Case - Control Panel Graph	40
What do we see?	40
Step 10 - Create a Campaign and Add Parameters	42
Add a Campaign	42
Adding Test Cases to the Campaign	43
Adding Project Parameters	43
Setting Parameters for a Campaign	45
Modify the Test Case to use Parameters	47
Programming the preCondition Function	48
Modifying the run Function to use Parameters	48
Programming the postCondition Function	49
Setting Parameters for Each Test Case	51
Step 11 - Execute the Campaign	53
What do we have?	53
Execution of a Campaign - Table View	53
Execution of a Campaign - Control Panel Graph	54
What do we see?	55
Step 12 - The Result Manager	56
The Result Manager	56
Step 13 - Test Results and Signal Traces	58
Test Results	58
Step 14 - The Log View and Log Levels	62
Log Levels	63

Log Levels in Test Cases	65
Step 15 - Test Reports	67
Test Reports	67

# Tutorial

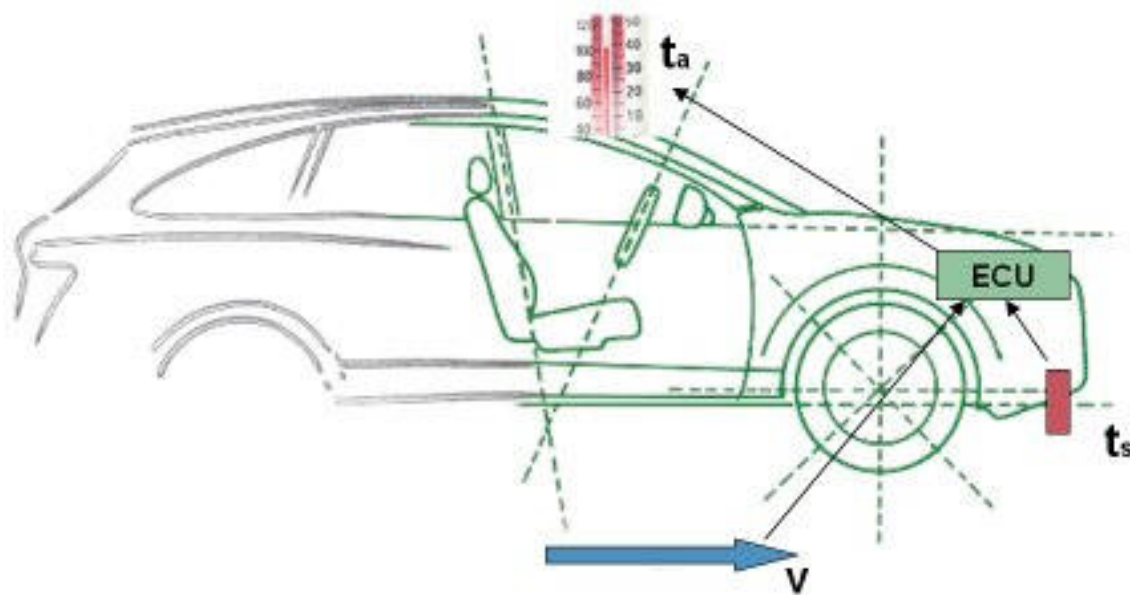
## Example: Temperature Compensation

The following sections will be used to create an example project to illustrate the various components of the MESSINA test development environment. Each step of the process will introduce and explain a MESSINA operation and will build on the previous steps. The test scenario is explained in detail below.

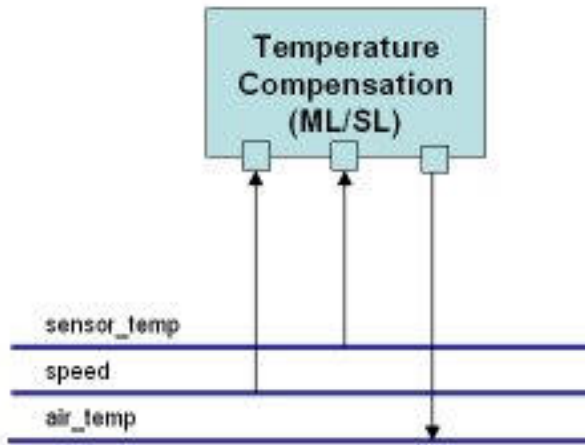
### Test Case Description

The sample scenario we will use is a temperature compensation for the outside car temperature. The temperature is acquired by a sensor and displayed to the driver. The measured temperature is affected by the speed of the car. To display the real outside temperature this effect must be compensated. An attenuation will be used to avoid big temperature steps. This calculation would normally be handled within the functionality of an ECU, but for purposes of illustration we will use a SiL (software in the loop) model.

The example project for the temperature compensation uses a MATLAB/Simulink model in place of the ECU functionality and three signals connected to the model's port.



The outside temperature signal from the sensor " $t_s$ " is recalculated depending on the vehicle speed ( $v$ ) before the actual temperature ( $t_a$ ) is displayed to the driver. The configuration in MESSINA is shown in the next figure:



We will use the MESSINA development environment to create and perform tests using the scenario described above. We will go through the process in small easy to follow steps giving full explanations as we go along. The following steps will be performed:

- setup and prepare the workspace
- create a new project
- add a model and test configuration
- add test signals
- add a target and run a first test
- add a table view and control panel visualization
- create and program a test case
- execute the test case
- add a campaign and parameters
- execute the campaign
- view the test results in the Result Manager
- examine traces and logging
- examine the Log View
- examine Test Reports

This example will use a Windows target which is installed automatically with MESSINA.

## Step 1 - Prepare the Test

Several operations must be completed before you can perform the tests described in the temperature compensation example. These are described in detail below.

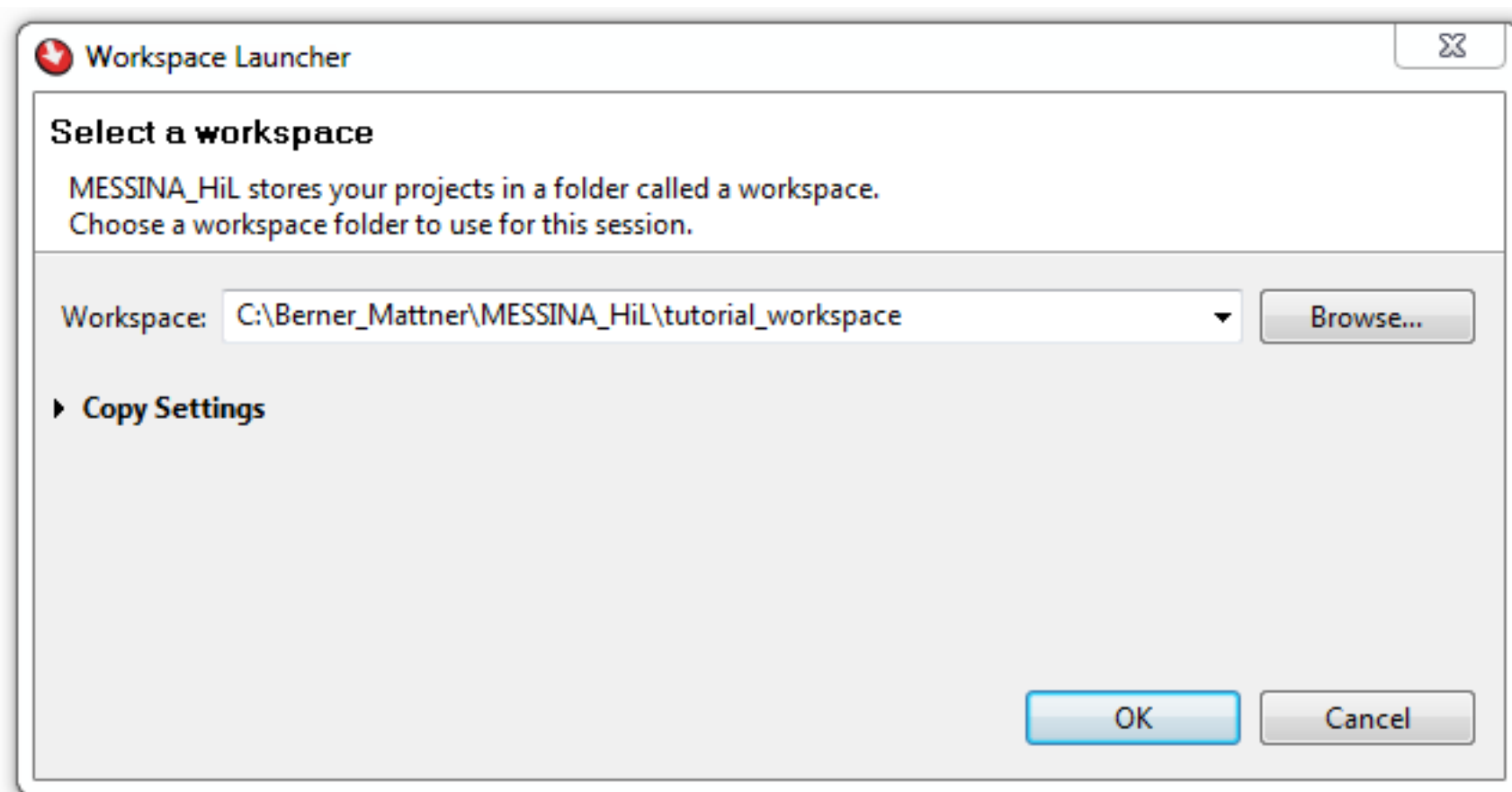
MESSINA uses the workspace sub-folder to create a structure for each project. The workspace sub-folder is located in the installation directory. A typical installation would be

***C:\Programme\Berner\_Mattner\MESSINA\workspace\ ....***

The project structure is created automatically and contains all information associated with a particular project.

This example will require that the MESSINA workspace be switched to the tutorial workspace which is part of the MESSINA installation.

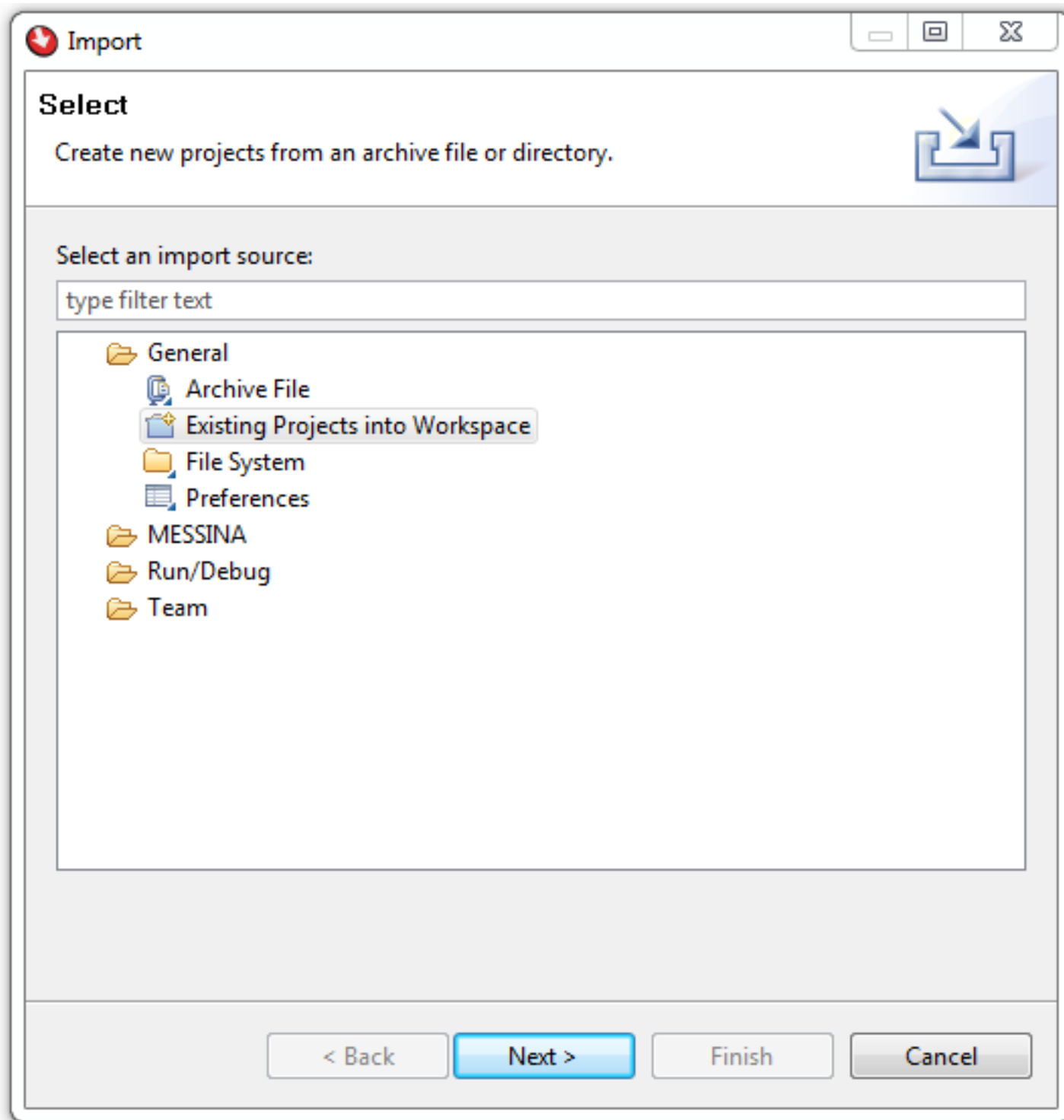
To switch the workspace open the dialog **File** → **Switch Workspace** → **Other** and select the **tutorial\_workspace** folder in the MESSINA installation directory.



**Note:** Changing the Workspace will restart MESSINA in the new Workspace.

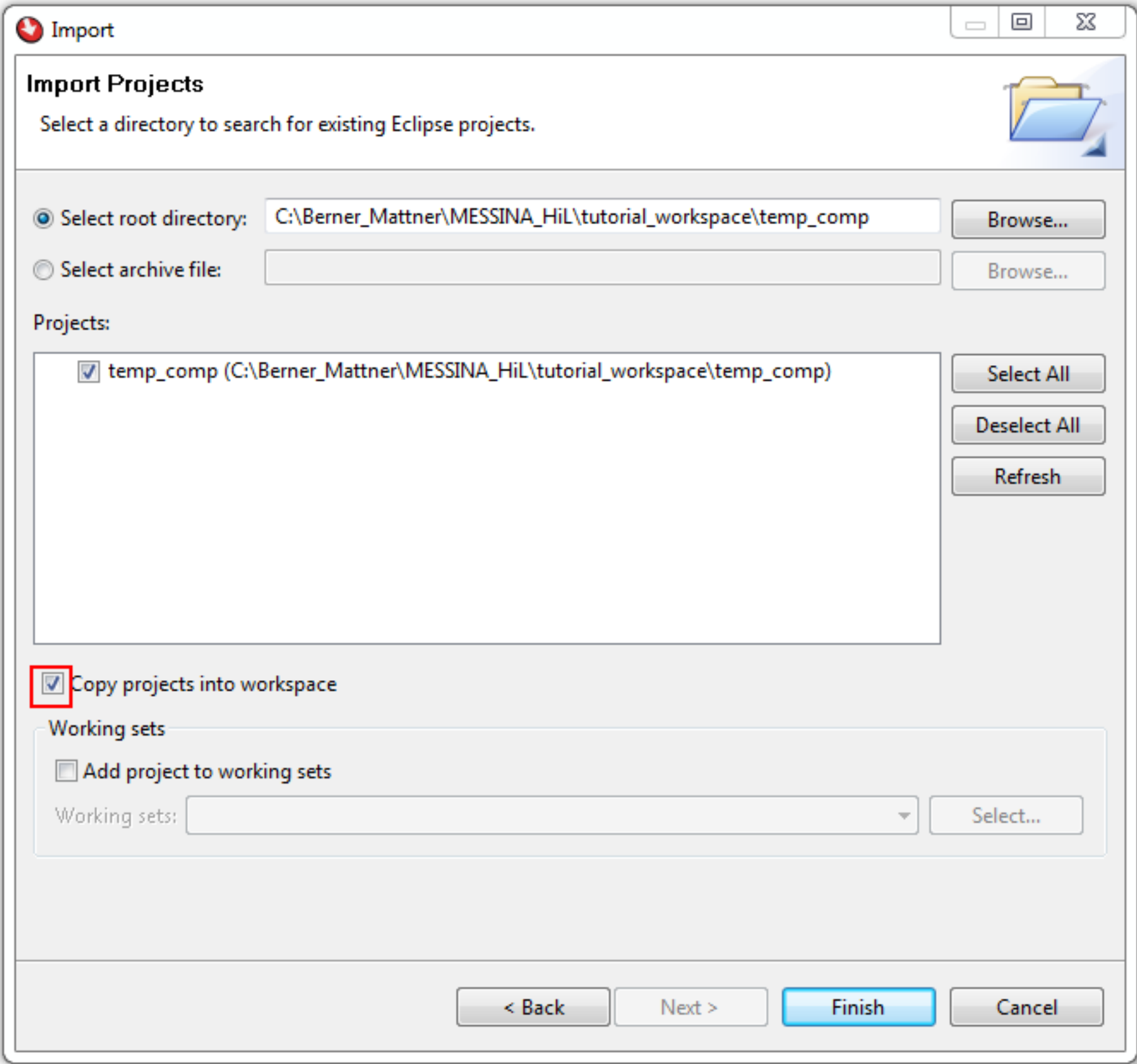
This tutorial workspace contains all information needed to finish the example. It also contains a MESSINA project with the completed example. To view this project you must first import the example project to your current workspace.

To import the project open the dialog **File** → **Import** and select **Existing Projects into Workspace** from the **General** folder and press **Next**.



Select the project **temp\_comp** in the folder **tutorial\_workspace** in the MESSINA installation folder and press the **Finish** button.

**Important Note:** Be sure that the **Copy project into workspace** checkbox is checked, otherwise an error will occur when the configuration is started.



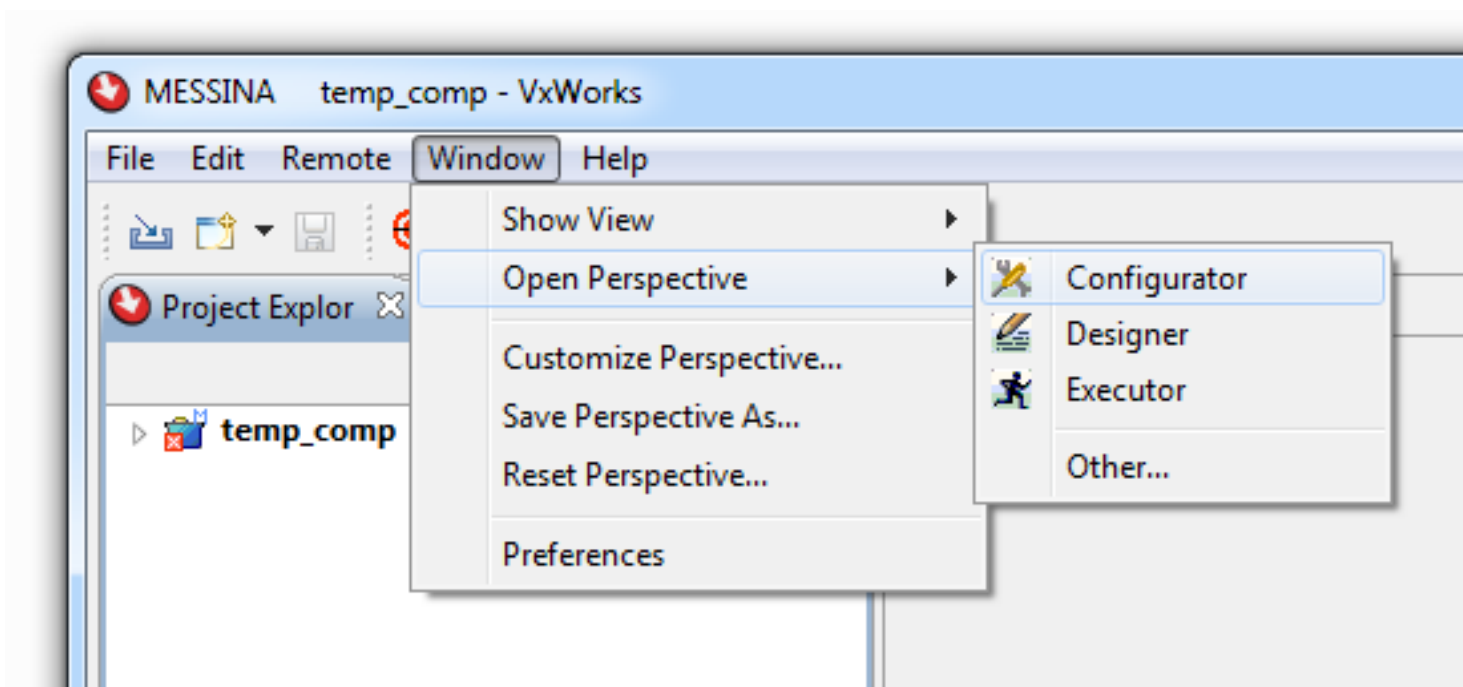
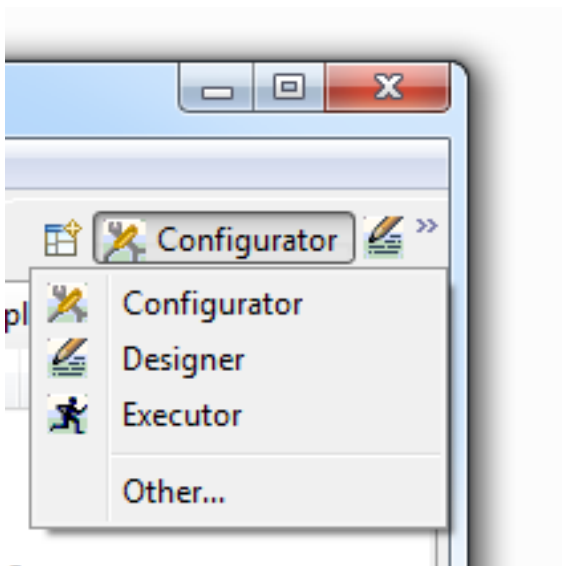
The project is now visible in your MESSINA workspace.



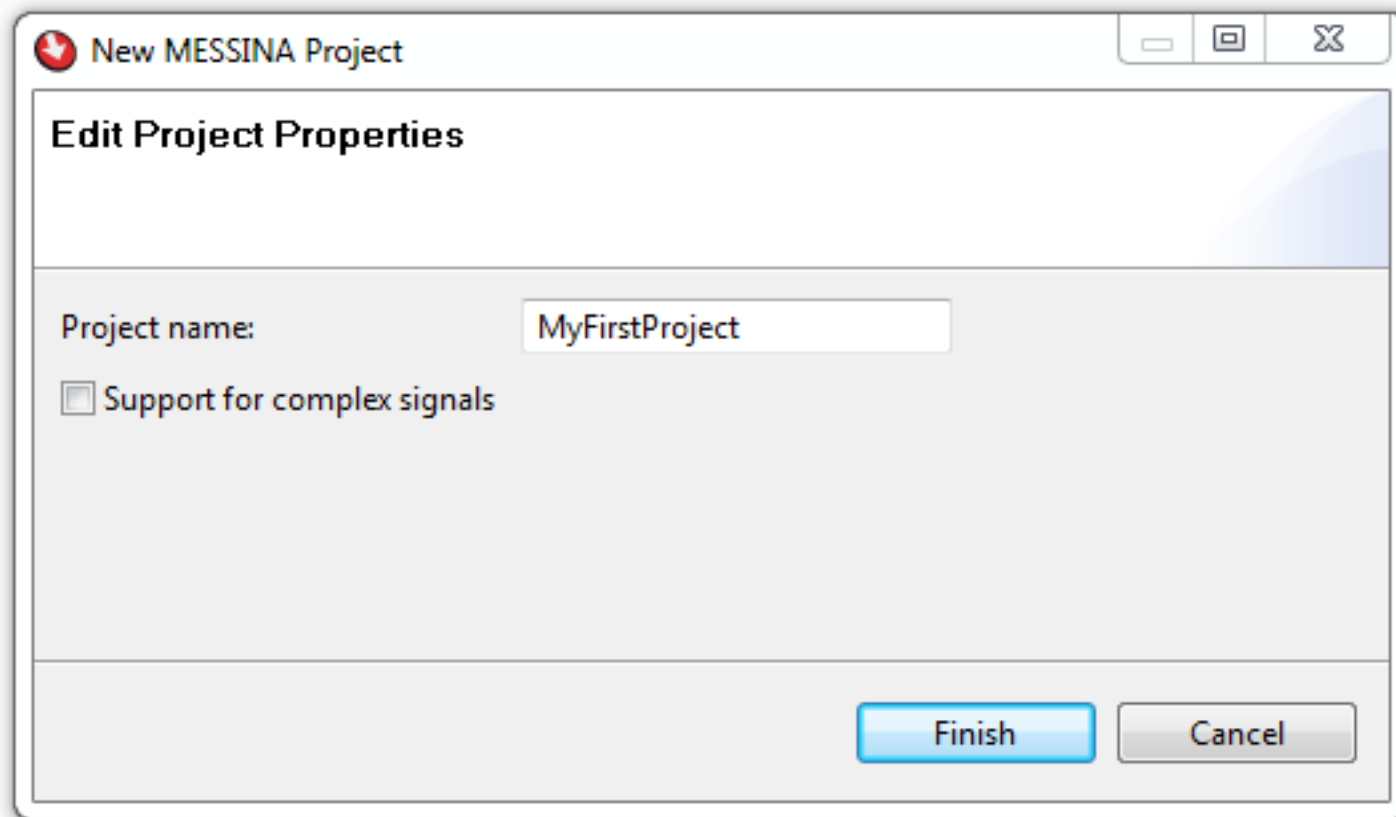
## Step 2 - Create New Project

This section will describe how to create a new MESSINA project. We will create a project to implement the temperature compensation example as previously described.

The MESSINA development environment has three standard **Perspectives**: **Designer**, **Configurator**, and **Executor**. Switching between these perspectives is done by selecting the appropriate tab at the top right of the main MESSINA window. Detailed descriptions of these perspectives are provided later in this documentation. For now, switch to the **Configurator** perspective to perform the following steps. This is done using **Main Menu** → **Window** → **Open Perspective** and selecting the **Configurator** as shown below.

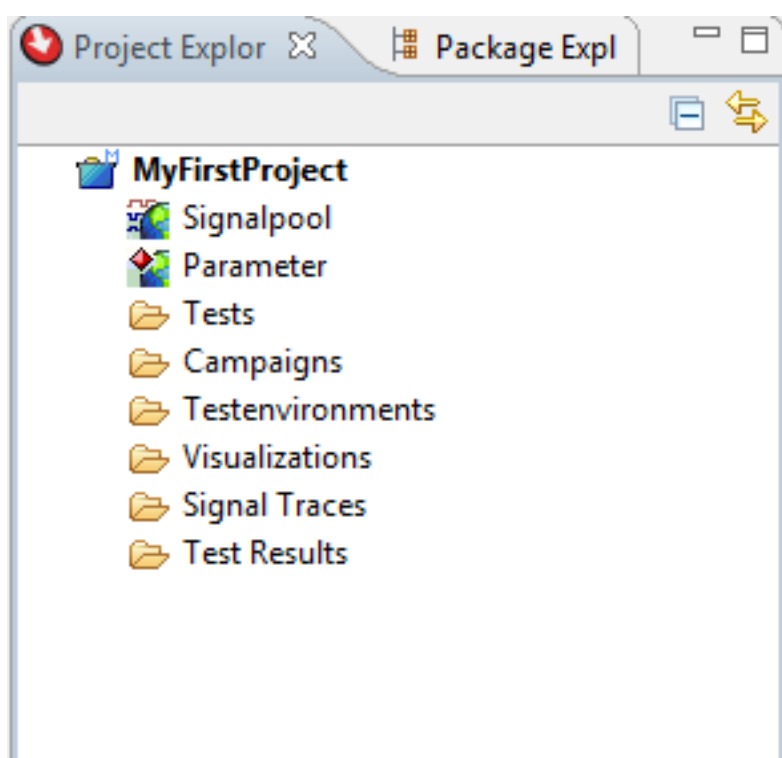


To create a new project open the dialog **File** → **New** → **MESSINA Project** and enter a name for the project (e.g. **MyFirstProject**).



Note the **Support for complex signals** checkbox available on the **New MESSINA Project** dialog box. Complex signals are either records (groups of signals) or arrays. If these types of signals are required in the project, this checkbox must be active. Our temperature compensation example does not require complex signals so the checkbox can remain unchecked.

Press **Finish** and a new project structure is shown in the [Project Explorer](#). All MESSINA projects use the same structure as shown below.



The first item in the project structure is the global **Signalpool**. The **Signalpool** is one of the most important features of the MESSINA test environment. It is an active process that runs on the target system. Each component ( e.g. hardware I/Os, models, test cases) is connected via the **Signalpool**. Each item in the **Signalpool** has a unique name, a unit, and a type (e.g. integer, float). Our temperature compensation example will require that we add several new signals to the **Signalpool**. These signals will be connected to a MATLAB/Simulink model.

The second item in the project structure is the **Parameter** item. **Parameters** are used like variables in a **Test Case** (test cases are described in detail below). **Parameters** are a powerful feature of MESSINA. They can be used to create variations of **Test Cases** by simply changing their value. This allows the test designer to create different Test Cases by using a standard test case and varying the **Parameters**.

The **Test Cases** folder in the project structure contains all available test cases which have been created in the project. A detailed explanation of **Test Cases**, what they are, and how to create and edit them is given in the **Using MESSINA → Windows & Views → Designer → [Test Case Editor](#)** section of this documentation. Our temperature compensation example will require that we add a test case to the project.

The **Campaigns** folder contains all available **Campaigns** which have been created in the project. A **Campaign** is a series of **Test Cases** used to perform specific tests in a defined order. The test cases are performed in the order they are listed in the **Campaign** folder.

The Testenvironments folder in the project structure contains the different configurations where the **Test Campaign** can run. These could be SiL (software in the loop), or HiL (hardware in the loop). For our temperature compensation example, we will create a SiL configuration which will be the **Testenvironments** used to run the test.

The **Visualizations** folder in the project structure contains the different visualizations that have been created in the project. A detailed explanation of **Visualizations**, what they are, the different types available, and how to create and edit them is given in the **Using MESSINA → Visualizations and Test Results → Visualizations** section of this documentation. Our temperature compensation example will require that we add visualizations to the project.

The **Signal Traces** folder in the project structure contains a list of all available signal trace files. A detailed explanation of signal traces, what they are, and how to create them is given in the **Using MESSINA → Visualizations and Test Results → Test Results → [Traces and Logging](#)** section of this documentation.

The **Test Results** folder in the project structure contains list of all available result files for the current project. A detailed explanation of **Test Results** is given in the **Using MESSINA → Visualizations and Test Results → Test Results → [Test Reports](#)** section of this documentation.

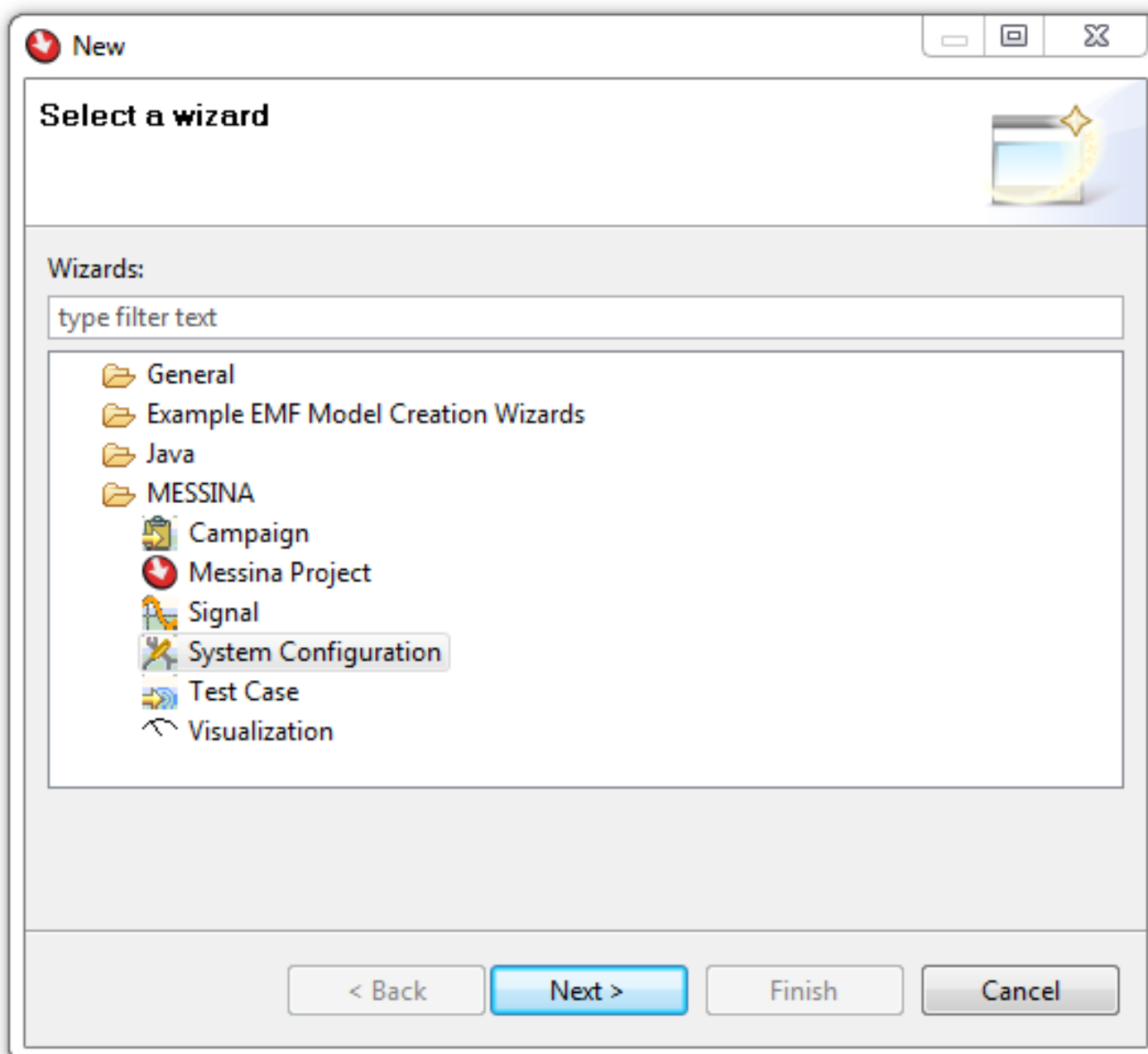
## Step 3 - Add Model and Configuration

This step shows how to add a system configuration to the **Testenvironments** of the project. We will also learn how to use the [Library Explorer](#) to **import** an existing Matlab/Simulink model into this configuration.

### Add a System Configuration to the Testenvironments

To add models, hardware and signals to the **Signalpool** we first must add a system configuration to the project. You can add more system configurations to handle different test scenarios like SiL or HiL.

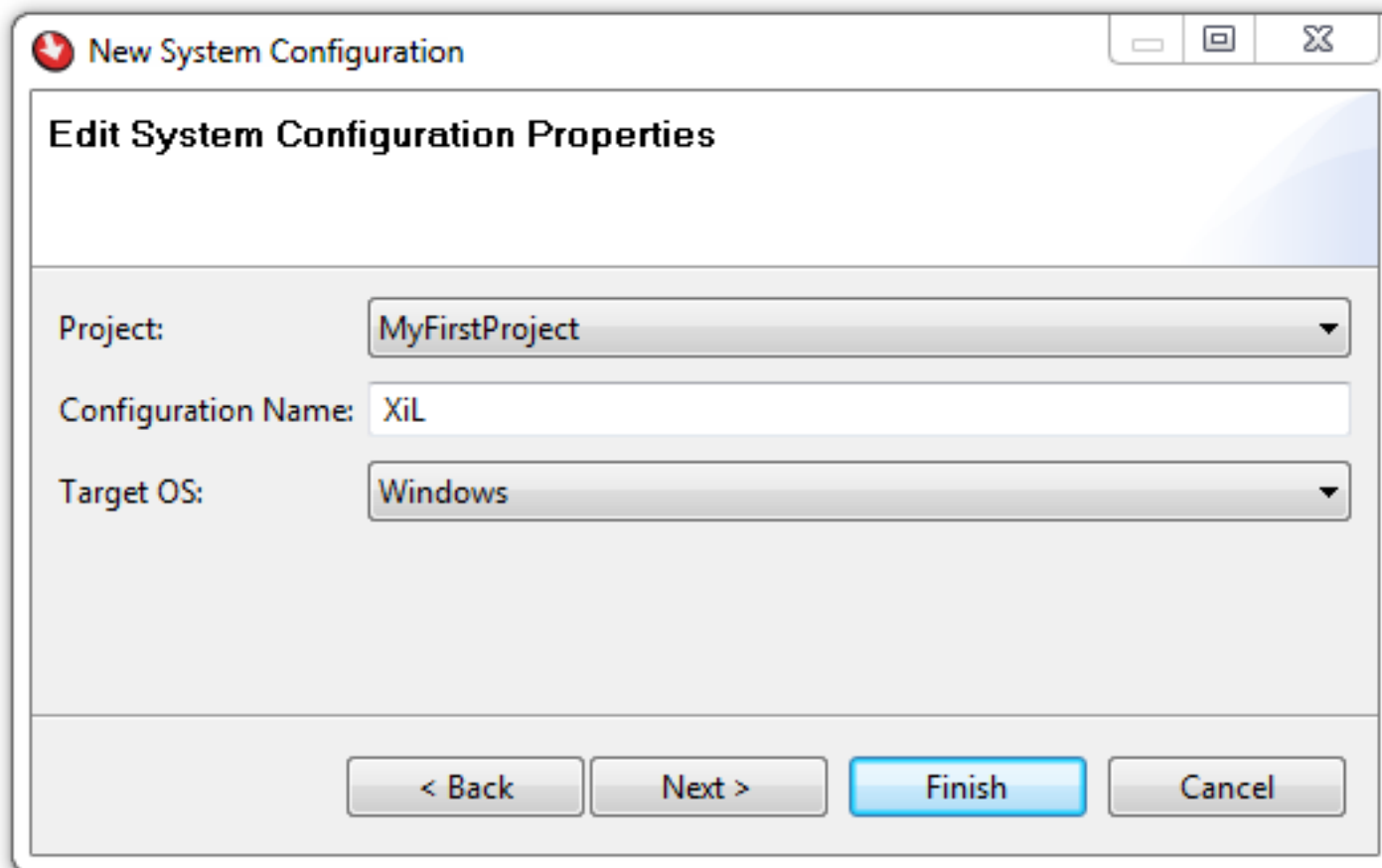
From the main menu, open the dialog **File** → **New** → **System Configuration**. It is also possible to open the **File** → **New** → **Other** dialog and select the **System Configuration** wizard in the MESSINA folder.



Alternatively, the **System Configuration Wizard** can be called using the **Context Menu** → **New Wizards** → **System Configuration**.

**Note:** When calling a wizard from the context menu, the desired wizard is not always shown directly. When this happens you must choose the **Other** option which opens the **Select a Wizard** dialog box shown above.

Enter the configuration name in the following dialog (e.g. XiL):

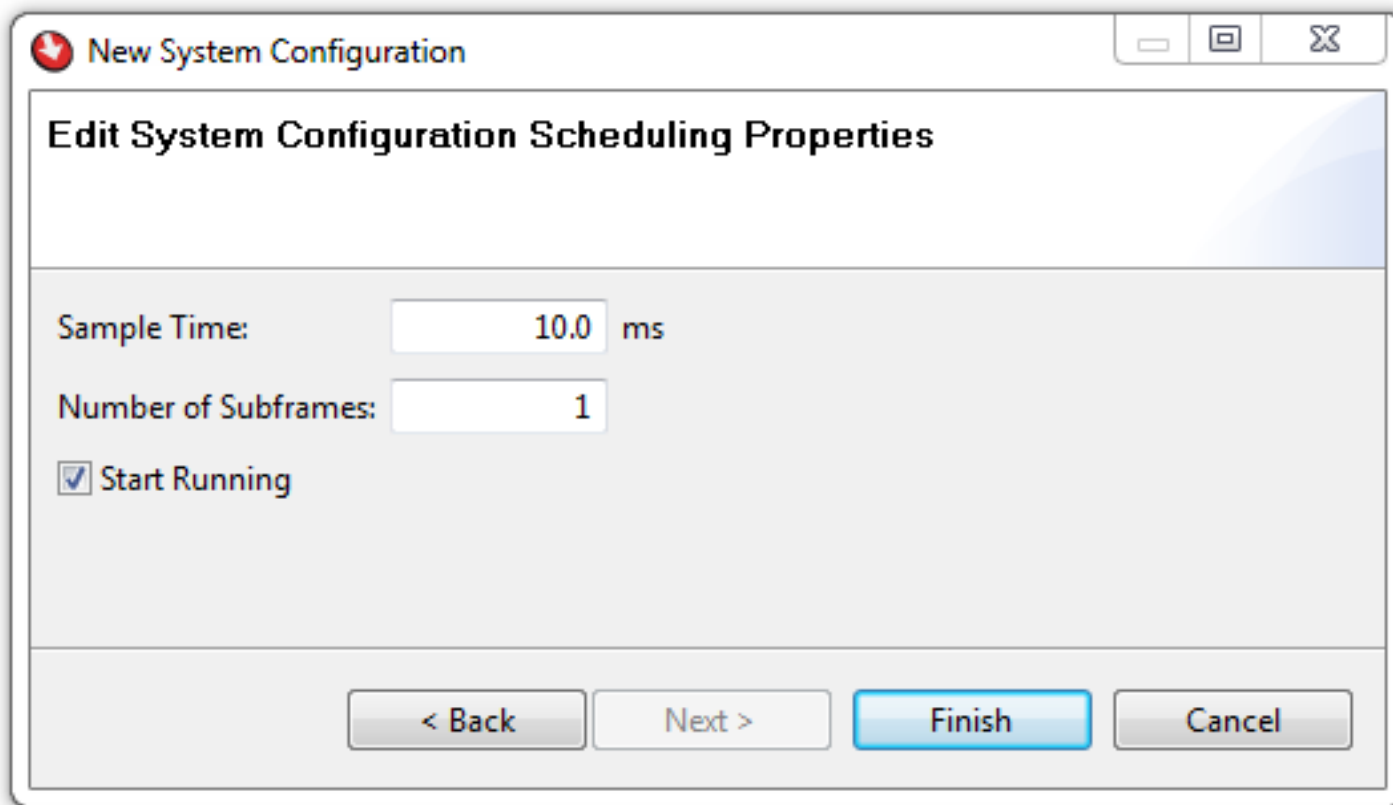


The image shows a Windows-style dialog box titled "New System Configuration". Inside the dialog, the title "Edit System Configuration Properties" is displayed. There are three input fields: "Project:" with a dropdown menu showing "MyFirstProject", "Configuration Name:" with a text box containing "XiL", and "Target OS:" with a dropdown menu showing "Windows". At the bottom, there are four buttons: "< Back", "Next >", "Finish" (highlighted in blue), and "Cancel".

The **Target OS** can be selected from the pull-down list. Currently supported target systems are Windows targets (installed automatically) and VxWorks targets (optional). For our example, select the Windows target.

**Note:** The components in the configuration must match the selected target system, in other words only Windows components can be used in a Windows target configuration and only VxWorks components can be used in a VxWorks target configuration.

Click on "**Next**" to edit the System Configuration Scheduling Properties.



By default the **Sample Time** is 10.0 ms for Windows target and 1.0 ms for VxWorks target. The **Number of Subframes** can be set to a number between 1 and 15.

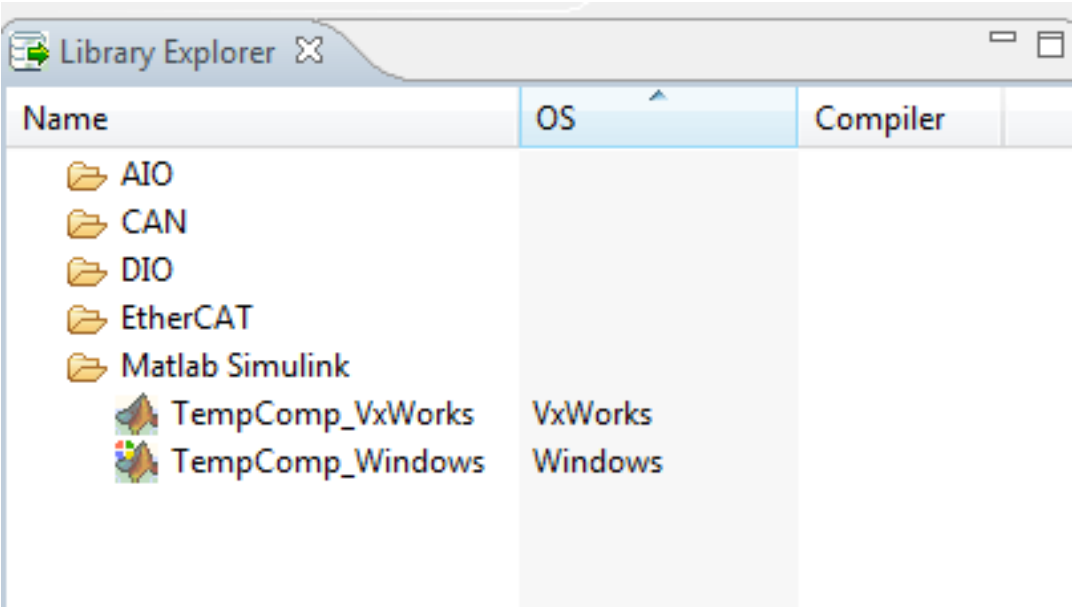
**Note:** Start Running is an upcoming feature and checked by default. To ensure a faultless function it has to remain activated.

Press "**Finish**" to close the dialog. The new system configuration is shown in the [Project Explorer](#) in the folder **Testenvironments**.

## Add a Matlab/Simulink Model to the Configuration

The ability to reuse existing items available through the [Library Explorer](#) is a powerful feature of MESSINA. Previous test models, models from earlier development and hardware configurations can all be added to the library and are then available for use in future projects. You can also create your own test models and add them to the library as required. The following picture shows the [Library Explorer](#) with all hardware items expanded.

**Note:** This section assumes that a MATLAB/Simulink model called "**TempComp**" already exists (it has been previously imported into the MESSINA library). This model is displayed in the [Library Explorer](#) view as shown in the following screen shot. Refer to the section **Import Library** for more details about importing a model into the MESSINA library.

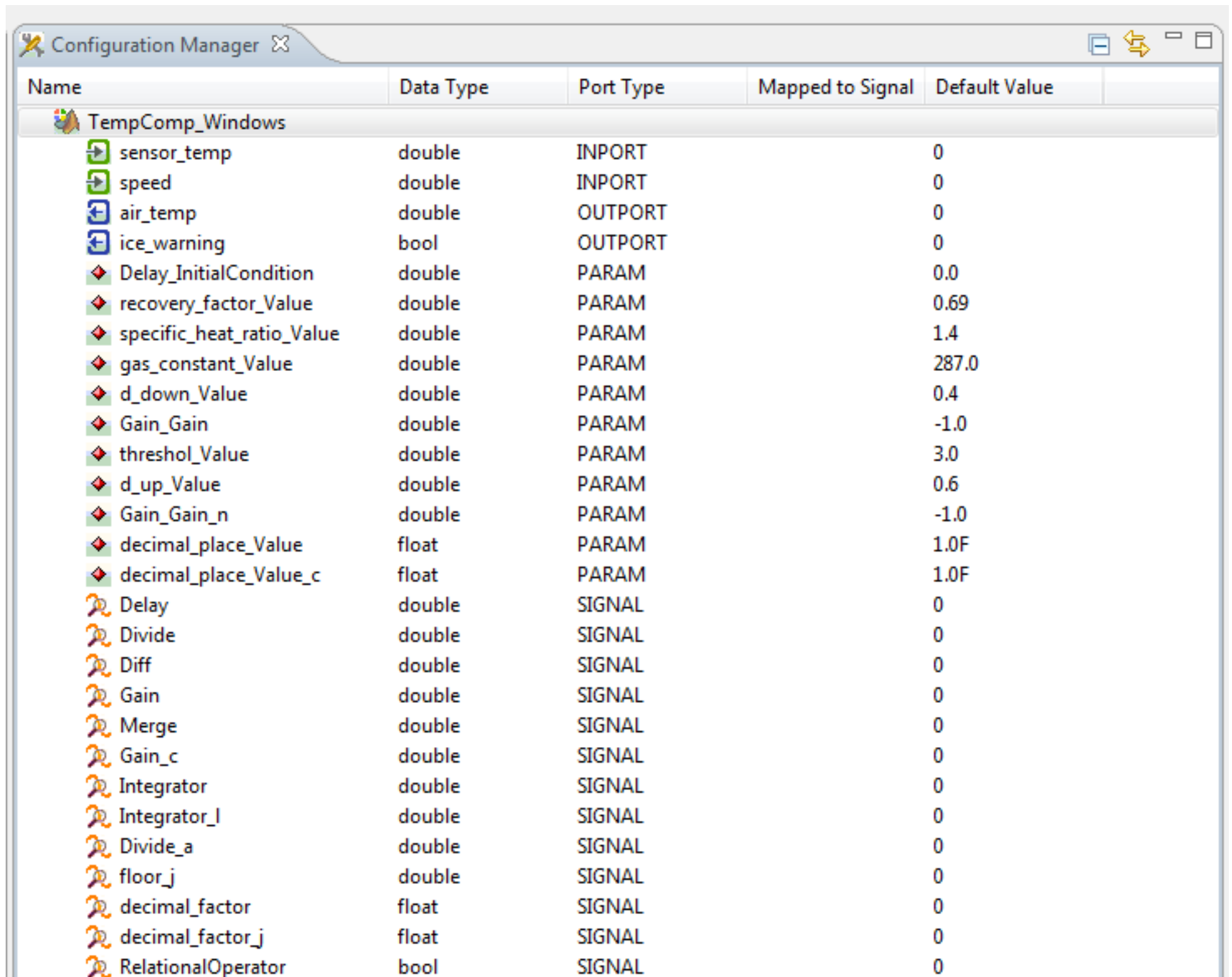


The MATLAB/Simulink model **TempComp** will be used in our temperature compensation example and must be added to our project. This model provides us with the functionality required to implement the temperature compensation operation as described in the previous section. The OS column indicates for which operating system the model was developed.

The **TempComp** model is added to the MESSINA configuration by drag-and-drop. To perform this action, you must be in the Configurator perspective. Mark the **TempComp** model in the [Library Explorer](#) and drag it to the MESSINA [Configuration Manager](#). Expand the **TempComp** item in the **Configuration Manager** to view all of the ports available in the model.

**Note:** The **Port Manager** is now integrated in the **Configuration Manager**. The properties of the model can be found in the **Properties View**.





Name	Data Type	Port Type	Mapped to Signal	Default Value
<b>TempComp_Windows</b>				
sensor_temp	double	INPORT		0
speed	double	INPORT		0
air_temp	double	OUTPORT		0
ice_warning	bool	OUTPORT		0
Delay_InitialCondition	double	PARAM		0.0
recovery_factor_Value	double	PARAM		0.69
specific_heat_ratio_Value	double	PARAM		1.4
gas_constant_Value	double	PARAM		287.0
d_down_Value	double	PARAM		0.4
Gain_Gain	double	PARAM		-1.0
threshol_Value	double	PARAM		3.0
d_up_Value	double	PARAM		0.6
Gain_Gain_n	double	PARAM		-1.0
decimal_place_Value	float	PARAM		1.0F
decimal_place_Value_c	float	PARAM		1.0F
Delay	double	SIGNAL		0
Divide	double	SIGNAL		0
Diff	double	SIGNAL		0
Gain	double	SIGNAL		0
Merge	double	SIGNAL		0
Gain_c	double	SIGNAL		0
Integrator	double	SIGNAL		0
Integrator_I	double	SIGNAL		0
Divide_a	double	SIGNAL		0
floor_j	double	SIGNAL		0
decimal_factor	float	SIGNAL		0
decimal_factor_j	float	SIGNAL		0
RelationalOperator	bool	SIGNAL		0

Notice that each port has an icon associated with it. These indicate the type of the port (can also be seen in the **Port Type** column).

The items listed in this view can be sorted by clicking on the column heading of the desired sort criteria. For example, to list the ports by **Data Type** order, click on the **Data Type** column header. The ports are sorted in ascending order by their data type. A small arrow in the column header indicates the sorting order (up arrow for ascending and down arrow for descending order).

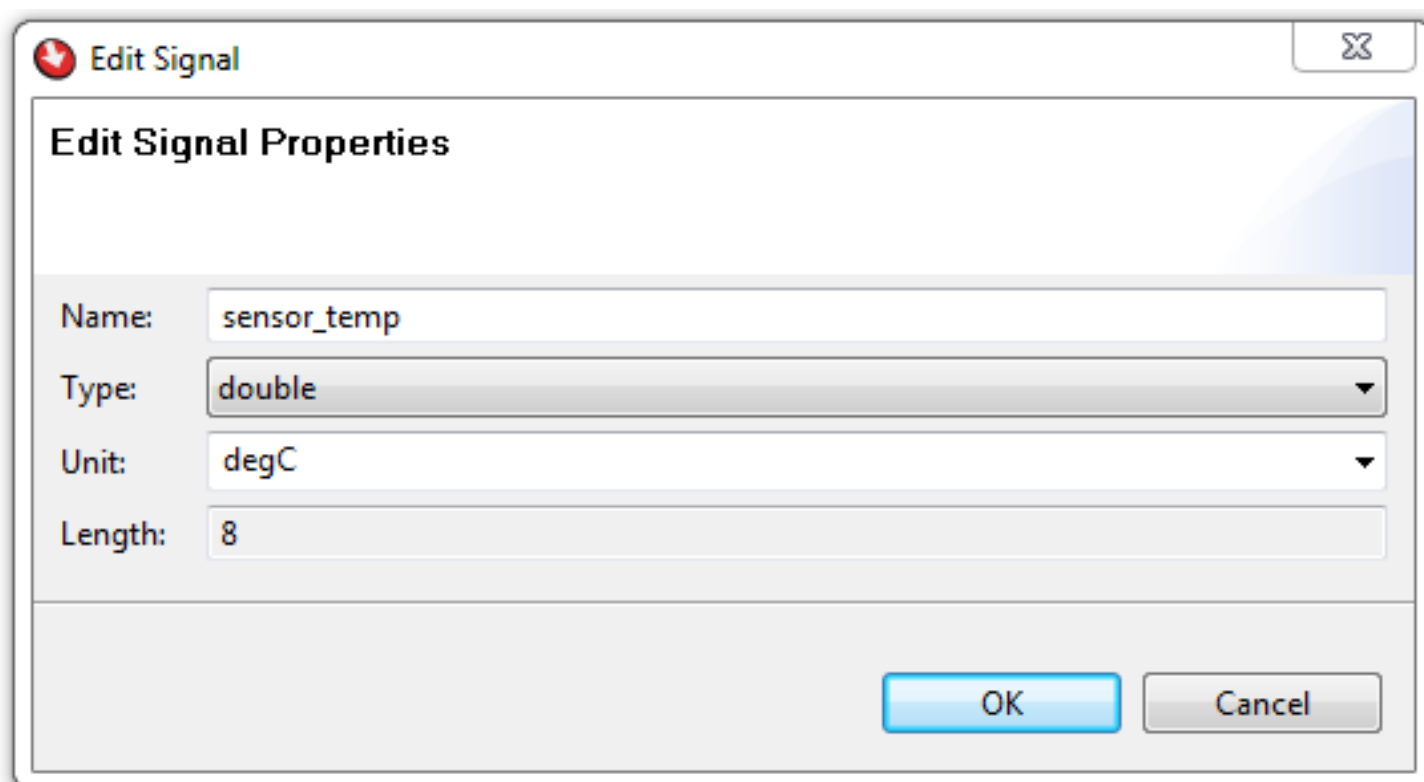


## Step 4 - Adding Signals to the Signalpool

The **Signalpool** is the central point where all MESSINA data is collected and distributed. It is an active process running on the target operating system. After creating a new MESSINA project, the **Signalpool** is empty. Signals that are required for visualization, monitoring, and evaluation must be added to the **Signalpool**. The **TempComp** model that we added in the previous step includes many different signals. The ones which are interesting for our temperature compensation example project are: **sensor\_temp**, **speed**, **air\_temp** and **ice\_warning**. Only these signals need to be added to the **Signalpool**.

### Add Signals to the Signalpool

Mark the **sensor\_temp** port in the expanded view of the **Configuration Manager**. Call the **Context Menu** → **Map To** → **New Signal**. The dialog box shown below is opened.



The pull-down lists for **Type** and **Unit** can be used to set the required properties for the signal. The signal name can be edited, too, if required. For our example, make the settings shown in the picture above (only the **Unit** must be changed).

**Note:** It is not required that the signal name assigned is identical with the port name. The name assigned here is how the signal will be displayed in the [Signalpool Manager](#).

If all properties are set, press the **OK** button. The new signal **sensor\_temp** is now shown in the **Signalpool Manager**. Notice that the icon assigned to the **sensor\_temp** signal in the **Configuration Manager** has now changed (an overlay icon was added). This indicates that the port has been mapped to the **Signalpool** and the name and ID of the connected signals are shown in the **Mapped to Signal**

column. Perform the same for the *speed*, *air\_temp*, and *ice\_warning* signals.

Configuration Manager

Name	Data Type	Port Type	Mapped to Signal	Default Value
TempComp_Windows				
sensor_temp	double	INPORT	sensor_temp (1)	0
speed	double	INPORT	speed (2)	0
air_temp	double	OUTPORT	air_temp (3)	0
ice_warning	bool	OUTPORT	ice_warning (4)	0
Delay_InitialCondition	double	PARAM		0.0
recovery_factor_Value	double	PARAM		0.69
specific_heat_ratio_Value	double	PARAM		1.4
gas_constant_Value	double	PARAM		287.0
d_down_Value	double	PARAM		0.4
Gain_Gain	double	PARAM		-1.0
threshol_Value	double	PARAM		3.0
d_up_Value	double	PARAM		0.6
Gain_Gain_n	double	PARAM		-1.0
decimal_place_Value	float	PARAM		1.0F
decimal_place_Value	float	PARAM		1.0F

Signalpool Manager

Name	Id	Length	Type	Unit	Path
Signalpo					
sensc 1		8	double	degC	
speec 2		8	double	km/h	
air_te 3		8	double	degC	
ice_w 4		1	bool		

The items listed in this view can be sorted by clicking on the column heading of the desired sort criteria. For example, to list the signals by *ID* order, click on the *ID* column header.

**Note:** There is no *Unit* for the *ice\_warning* signal because it is a simple boolean value.


# Step 5 - Add, Configure and Start a Target

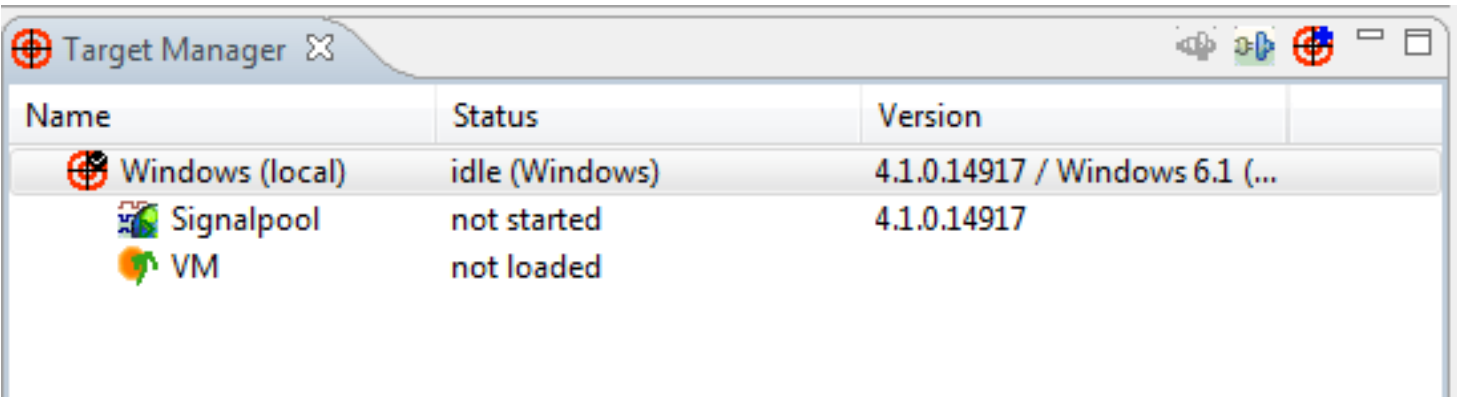
The MESSINA installation always includes a Windows target. In order to run our test, we need to connect to a target system. We require a target system with a known IP address to be connected to our host system. Target Manager settings are done in the **Executor** perspective, so switch to this perspective using the tabs in the top right corner of the main MESSINA view.

## Add and Configure a Target

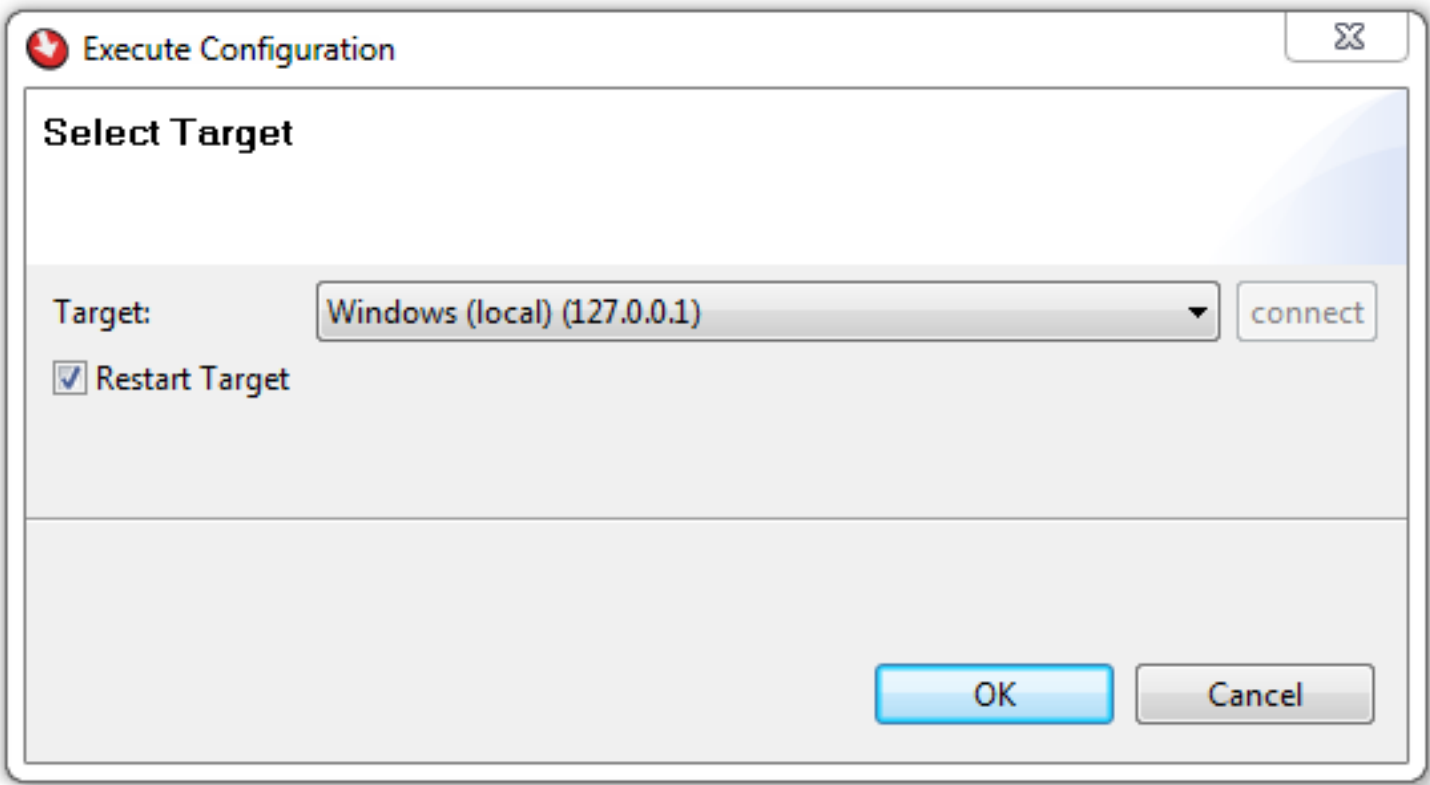
The Windows target is always included in the MESSINA installation. Refer to the [Target Manager](#) section of this documentation for detailed information about how to add and configure a target. Make sure that the Windows target shown is set as the default target. This can be done by selecting the target, then calling the **Context Menu** → **Set as Default Target**. A small overlay icon with a check

mark  indicates the currently selected default target.

The target can be connected either by pressing the icon  or by calling the **Context Menu** → **connect**. The **Target Manager** now looks like the picture below:



One final action is required before the entire process starts. Go back to **Project Explorer** → **Testenvironments** → **XiL**. The configuration must be started using the **Context Menu** → **Start Configuration** option. The following dialog is called:

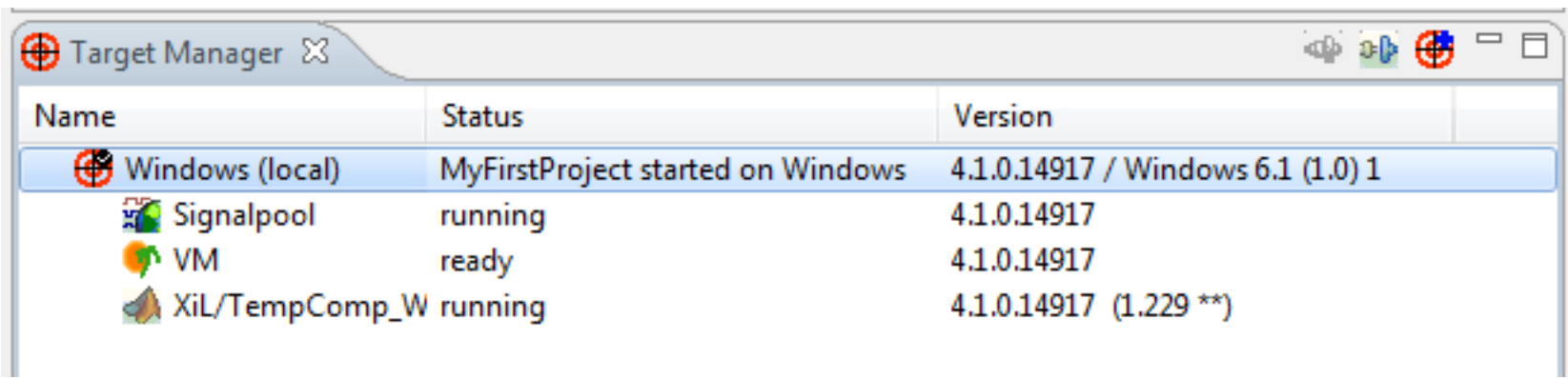


The Target selected from the pull-down list (all available targets are listed) must match the current target. The **Restart Target** check box can be left active. Press **OK** to perform the start configuration action.

**Note:** **Restart Target** does NOT re-boot the VxWorks operating system. Activating this check-box will restart the model (**TempComp**) and the **Signalpool** process running on the target system.

The **Start Configuration** dialog is called and disappears automatically when the configuration is completed.

If everything has been connected and started properly, the [Target Manager](#) display will now look like the screenshot below:



Notice that the **Signalpool** and **TempComp** objects have a **Status** of **running**. Both of these items are active processes that run directly on the target system after the **Configuration** is started. In the next steps we will have a closer look at how the model works and what the results of our [Test Case](#) look like.

## Step 6 - Add a Visualization and Test Model

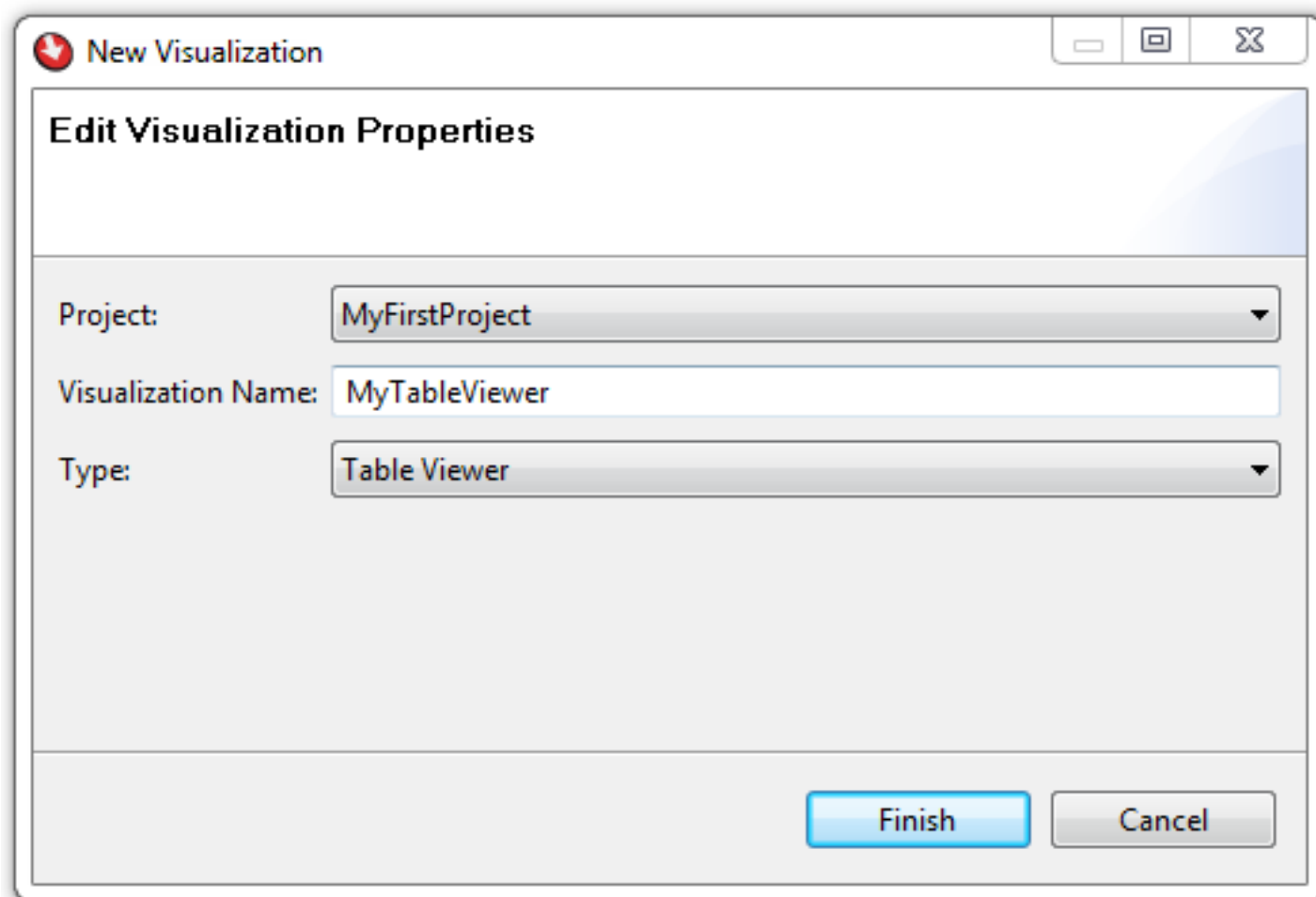
The previous steps in this tutorial described how to add what we need in order to run a test. Adding a **Visualization** to the project will allow us to see what is happening during execution. MESSINA offers many different types of **Visualizations** for viewing selected signals. Any signal in the **Signalpool** can be added to a **Visualization**. For our temperature compensation example, we will use a [TableView](#) **Visualization** to monitor the different signals that we added earlier in this tutorial.

### Add a Table Viewer Visualization

MESSINA **Visualizations** are created and edited using the **Executor** perspective which can be selected using the tabs at the top right of the main window. From the main menu, open the dialog **File** → **New** → **Visualization**. It is also possible to open the **File** → **New** → **Other** dialog and select the Visualization wizard in the MESSINA folder.

Alternatively, the **Visualization Wizard** can be called using the **Context Menu** → **New Wizards** → **Visualization**.

The following dialog box appears when the **Visualization Wizard** is called:



The **Project** pull-down list can be used to select the project where the **Visualization** is to be created. All projects listed in the [Project Explorer](#) will be available in the pull-down list. The currently active project is automatically set as the selected item.

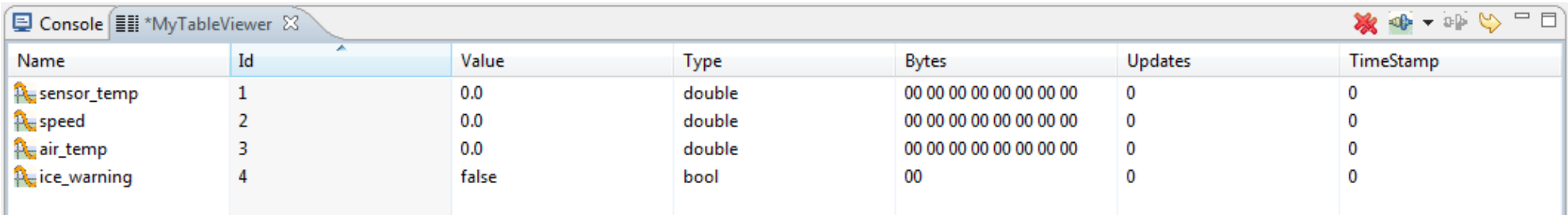
The **Visualization Name** text box is used to enter the **Visualization** name. A sample name is always generated automatically, but this can be changed to any **Visualization** name that does not already exist in the project.

**Note:** Blank spaces are not allowed in the name.

The **Type** pull-down list can be used to select what type of **Visualization** will be added to the project. Select the **Table Viewer visualization**. A detailed description of all visualizations can be found in the **Using MESSINA → Visualizations and Test Results → Visualizations → Table Displays** section of this documentation.

Enter **MyTableView** and press **Finish** to create the **Visualization**.

The newly created **Visualization** is added to the **Executor** view, but it is empty (there are no signals to view). Signals are added to the **visualization** using drag-and-drop. Mark the **sensor\_temp** signal (either in the [Project Explorer](#) or the [Signalpool Manager](#)) and drag it to the **Visualization** view. Do the same for the **speed**, **air\_temp**, and **ice\_warning** signals. The **Visualization** view should now look like this:



Name	Id	Value	Type	Bytes	Updates	TimeStamp
sensor_temp	1	0.0	double	00 00 00 00 00 00 00 00	0	0
speed	2	0.0	double	00 00 00 00 00 00 00 00	0	0
air_temp	3	0.0	double	00 00 00 00 00 00 00 00	0	0
ice_warning	4	false	bool	00	0	0

The items listed in this view can be sorted by clicking on the column heading of the desired sort criteria. For example, to list the signals by **ID** order, click on the **ID** column header.

## Testing the TempComp Model

We can examine the operation of the **TempComp** model by directly modifying the **speed** and

**sensor\_temp** input signals. Connect the **Table View visualization** by pressing the  icon. Make sure the target is connected, and then start the configuration as we did in the last step.

Double click the **sensor\_temp** signal in the **MyTableView** view (**Table Display** visualization) created earlier. The following dialog box appears:

Signal Value

Set New Signal Value

Name:

sensor\_temp

Old Value:

0.0

New Value:

40.0

OK

Cancel

Enter a **New Value** (e.g. 40.0 as shown) and press the **OK** button. Wait until the **air\_temp** signal reaches 40.0. Then double click the **speed** signal and enter another **New Value** (e.g. 95) and press the **OK** button. The **Table View** visualization should now look like this:

Name	Id	Value	Type	Bytes	Updates	TimeStamp
sensor_temp	1	40.0	double	40 44 00 00 00 00 00 00	3	95870
speed	2	95.0	double	40 57 c0 00 00 00 00 00	3	137970
air_temp	3	36.9	double	40 42 73 33 33 33 33 33	370	157380
ice_warning	4	false	bool	00	3	100980

Notice that the **air\_temp** value is not the same as the **sensor\_temp** value. The model has compensated the **air\_temp** value based on the **speed** value that we entered earlier. Try entering different speed values to confirm that the compensation is working. A higher compensation effect is visible for higher speeds and a lower compensation effect at lower speeds.

Our temperature compensation example is now functioning properly. In the following section we will show you another way to view the operation of the **TempComp** model.



## Step 7 - Add a Control Panel Visualization

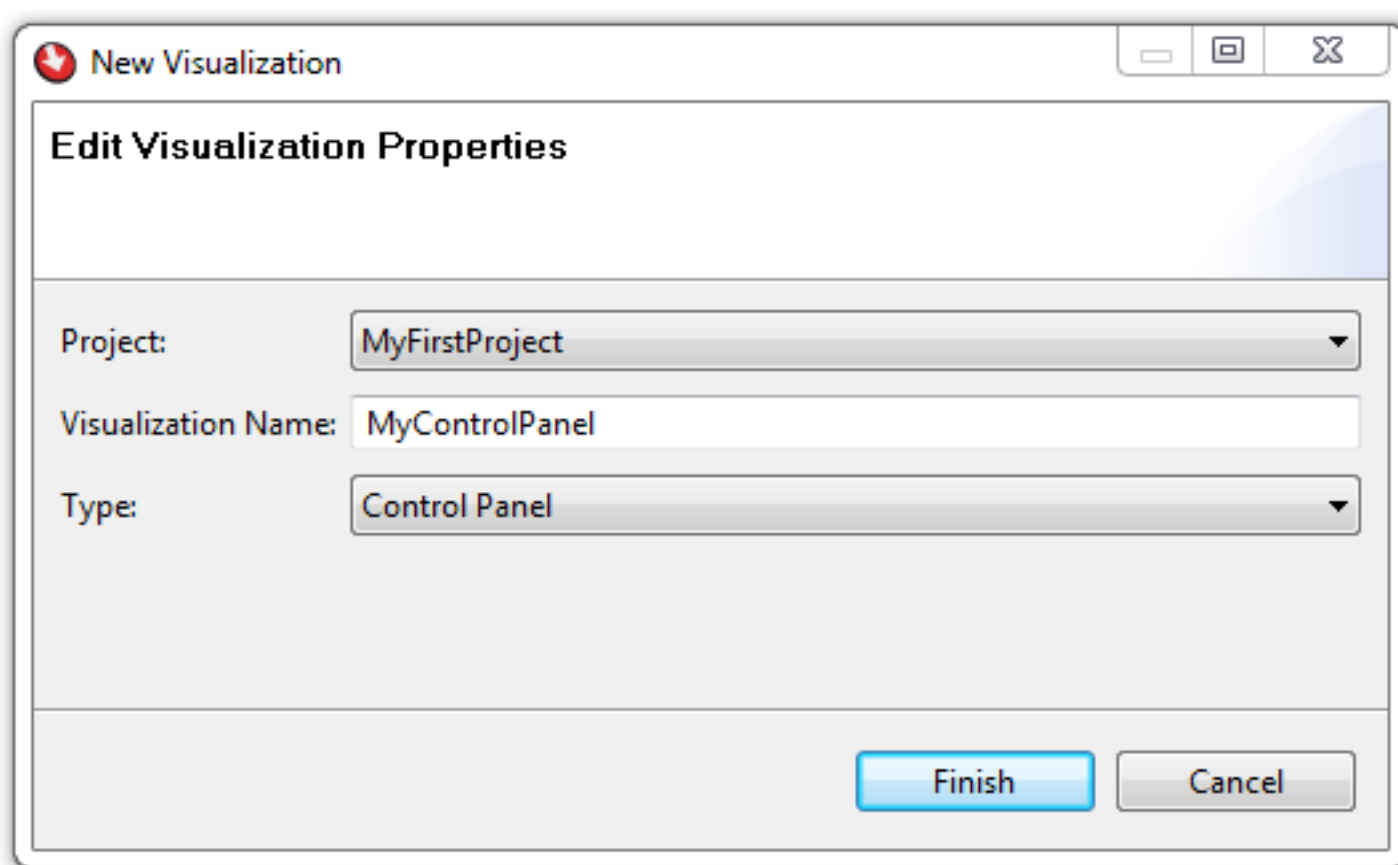
In this section we will take a close look at the **Control Panel visualization**. We will use this to attach to the signals of the MATLAB/Simulink model and use some of the elements available to control the operation of the model. We will run the model and use the control panel elements that we will add to see how the model reacts. For a detailed description of **Control Panels**, refer to the **Using MESSINA** → **Views** → **Visualizations and Test Results** → **Visualizations** → [Control Panel](#) section of this documentation.

### Add a Control Panel Visualization

MESSINA **Visualizations** are created and edited using the **Executor** perspective which can be selected using the tabs at the top right of the main window. From the main menu, open the dialog **File** → **New** → **Visualization**. It is also possible to open the **File** → **New** → **Other** dialog and select the **Visualization Wizard** in the MESSINA folder.

Alternatively, the **Visualization Wizard** can be called using the **Context Menu** → **New Wizards** → **Visualization**.

The following dialog box appears when the **Visualization Wizard** is called:

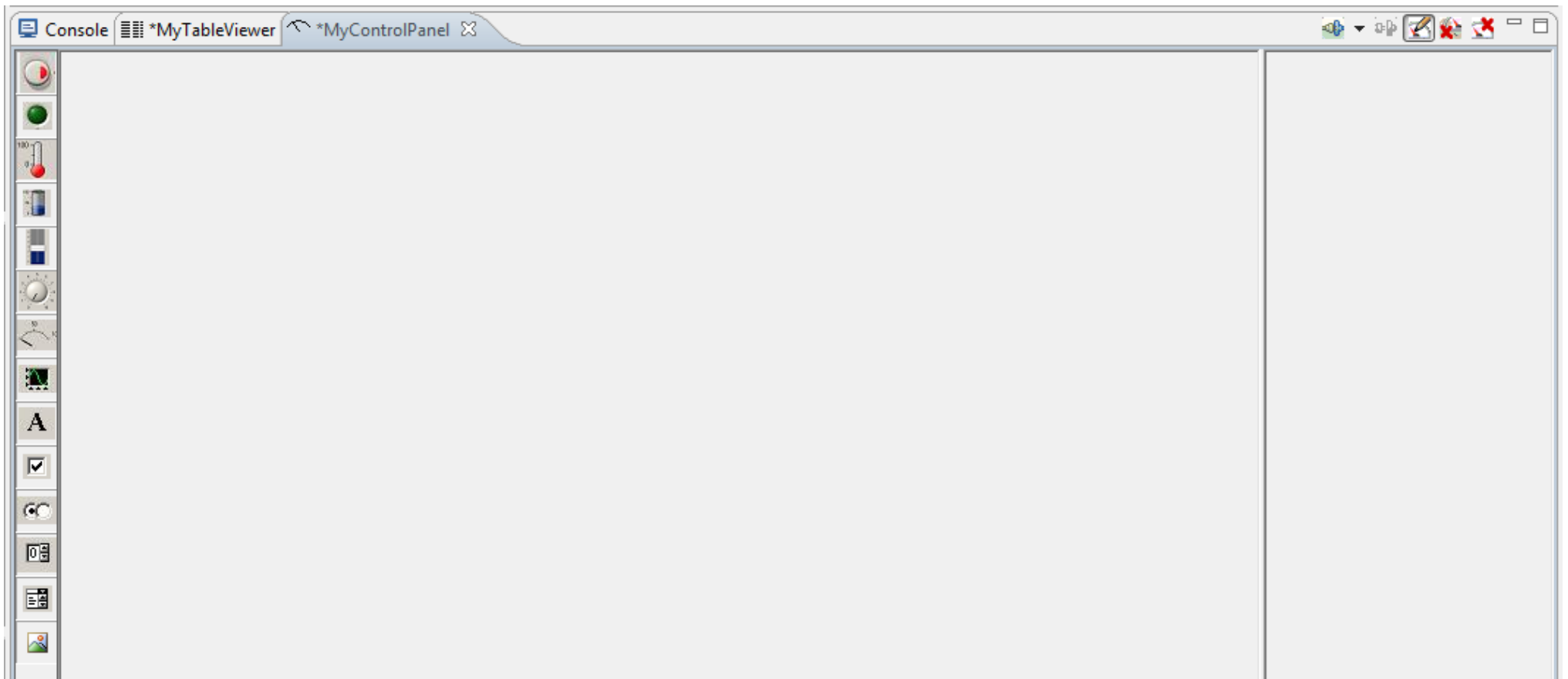


Enter **MyControlPanel** and press **Finish** to create the **Visualization**.

The newly created **Visualization** is shown, but it is empty. Press the  icon at the top of the



visualization view to enter the **Edit Mode**. The view should now look like this:



This visualization gives you the ability to design your own control panel. The available elements are listed down the left side of the view. Our goal here is to select the elements we need in order to do the following:

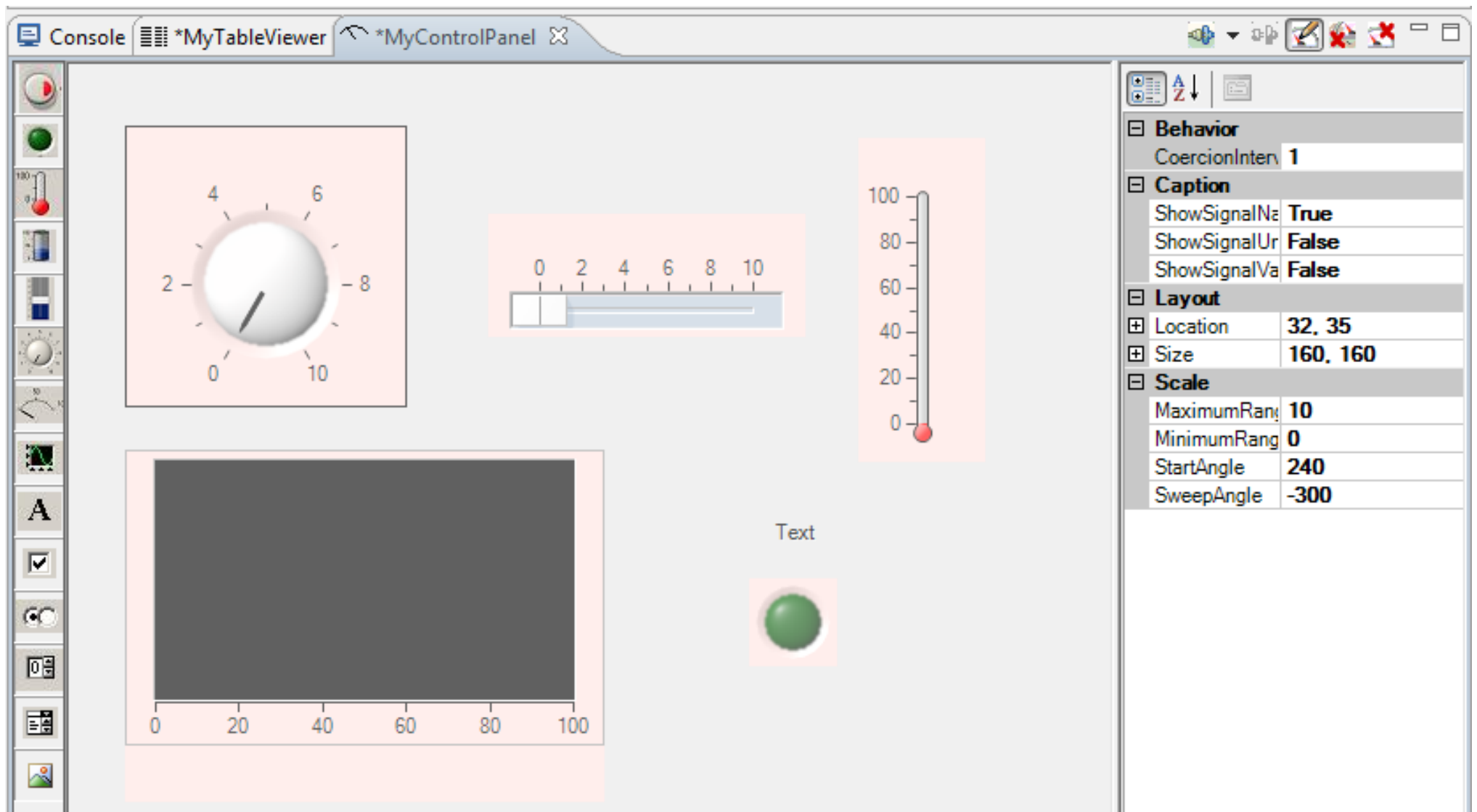
- change the value of the speed signal
- change the value of the sensor\_temp signal
- display the air\_temp signal
- display the ice\_warning signal
- plot a graph of air\_temp vs speed over time

In other words what we did in the previous step using the [Table View](#) visualization is what we want to do here with [Control Panel](#) elements.

Add the following elements to the [Control Panel](#) by selecting them from the list on the left and dragging them into the panel area:

- a knob (dial) for setting the speed
- a slider for setting the sensor\_temp
- a thermometer for displaying the air\_temp
- an LED for displaying the ice\_warning
- a graph for plotting air\_temp vs speed over time

Arrange and size the elements to suit your taste. An example is shown below (in **Edit Mode**):



The display elements have a pink background in the **Edit Mode**. This indicates that the control has not been connected to a signal. The section on the right is used to set properties which apply to the currently selected display element. Each element has a different list of properties. The above properties are for the dial element which is selected. We want to use the dial to set the **speed** signal.

Make the following settings for the dial:

- ShowSignalName: True
- ShowSignalUnit: True
- ShowSignalValue: True
- MaximumRange: 140
- MinimumRange: 0

Make the following settings for the slider:

- ShowSignalName: True
- ShowSignalUnit: True
- ShowSignalValue: True
- MaximumRange: 20
- MinimumRange: -10

Make the following settings for the thermometer:

- ShowSignalName: True
- ShowSignalUnit: True

- ShowSignalValue: True
- MaximumRange: 20
- MinimumRange: -10

Click on the LED. Set the ON and OFF colours to suite your taste. Remember, when the LED is ON, this will indicate that the **ice\_warning** is ON, so consider this when choosing the colours.


No changes are required for the graph element.

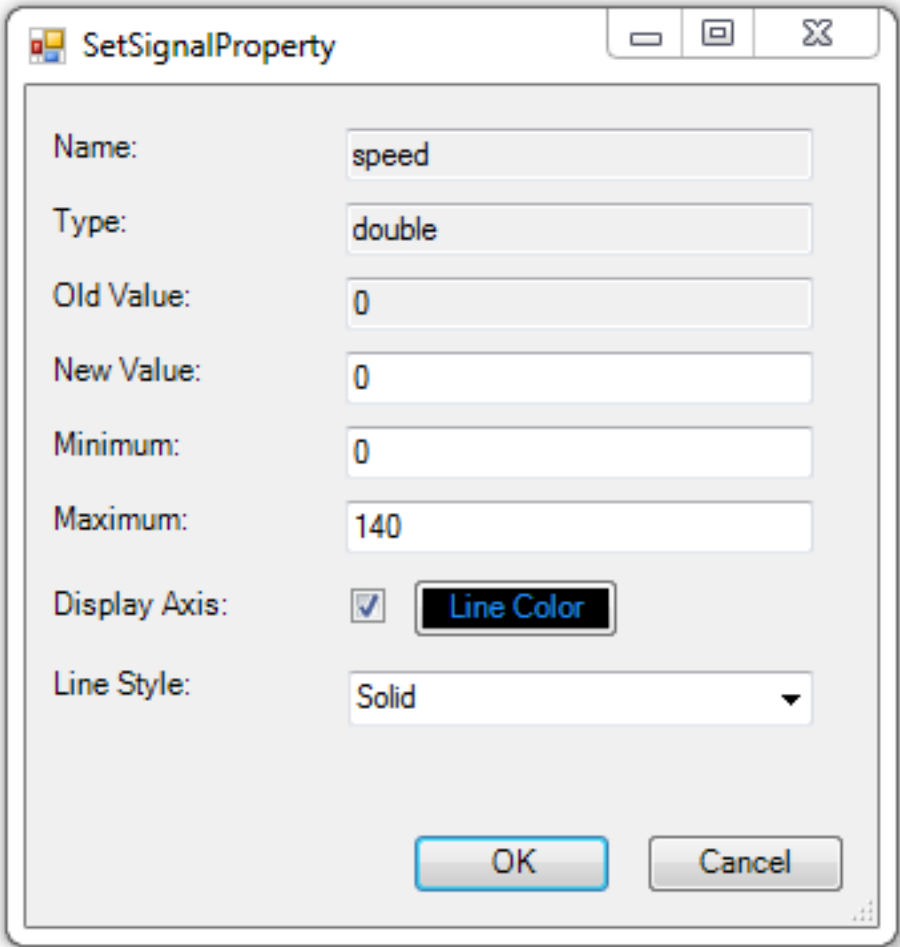
## Connect Signals to the Elements

We now need to add signals to the elements so that they can be used to control the desired signals of the **TempComp** model. Do this by marking the desired signal in [Signalpool Manager](#) view and dragging it on to the desired element as follows:

- drag the speed signal to the dial
- drag the sensor\_temp signal to the slider
- drag the air\_temp signal to the thermometer
- drag the ice\_warning signal to the LED
- drag the air\_temp signal to the graph
- drag the speed signal to the graph

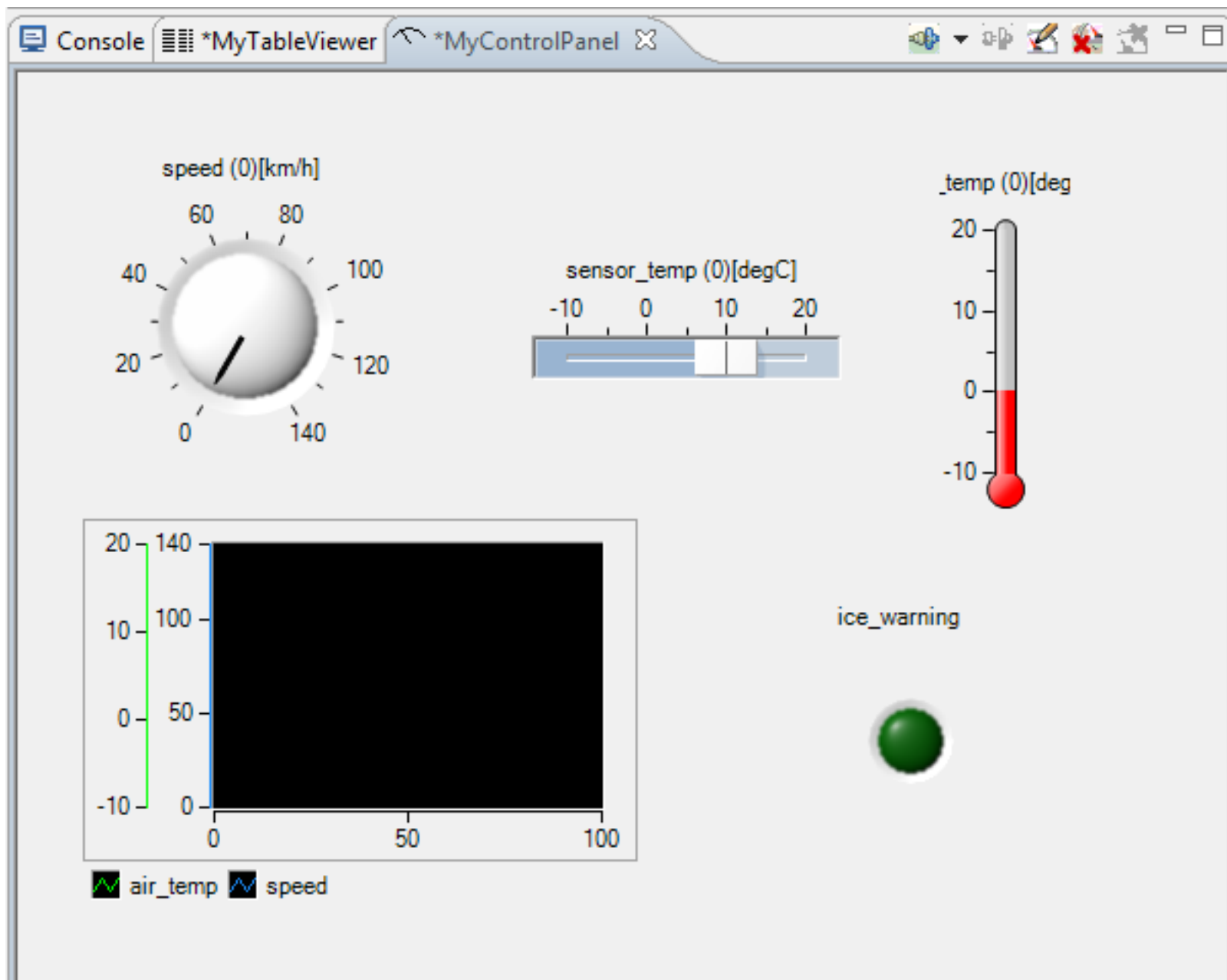
Note that the LED does not display a text. Add a label by dragging it from the element list on the left and placing directly above the LED. Change the text setting to read "ice warning". Alternatively you can just drag and drop a signal to the label to show its name.

Press the  icon at the top of the visualization view to exit the **Edit Mode**. One final setting is required before we can test our control panel. We will set the scale on the graph (this is not done in **Edit Mode**). Double click on the **speed** signal in the legend under the graph display. The following dialog appears:



Set the minimum and maximum values to those used for the dial (0 and 140). Double click on the ***air\_temp*** signal in the legend under the graph display and set it to -10 and 20 (the same as the other temperature display elements).

An example is shown below:



## Testing the TempComp Model

In the previous step we tested the model using the [Table View Display](#). We will now do the same using the panel we just finished creating. Connect the **Control Panel visualization** by pressing the

 icon. Make sure the target is connected, and then start the configuration as we did in the last step.

Take a bit of time to set different **speed** and **sensor\_temp** values with the controls. Examine the graph, the **air\_temp**, and the **ice\_warning** elements to convince yourself that the model is functioning properly.

## Step 8 - Create and Program a Test Case

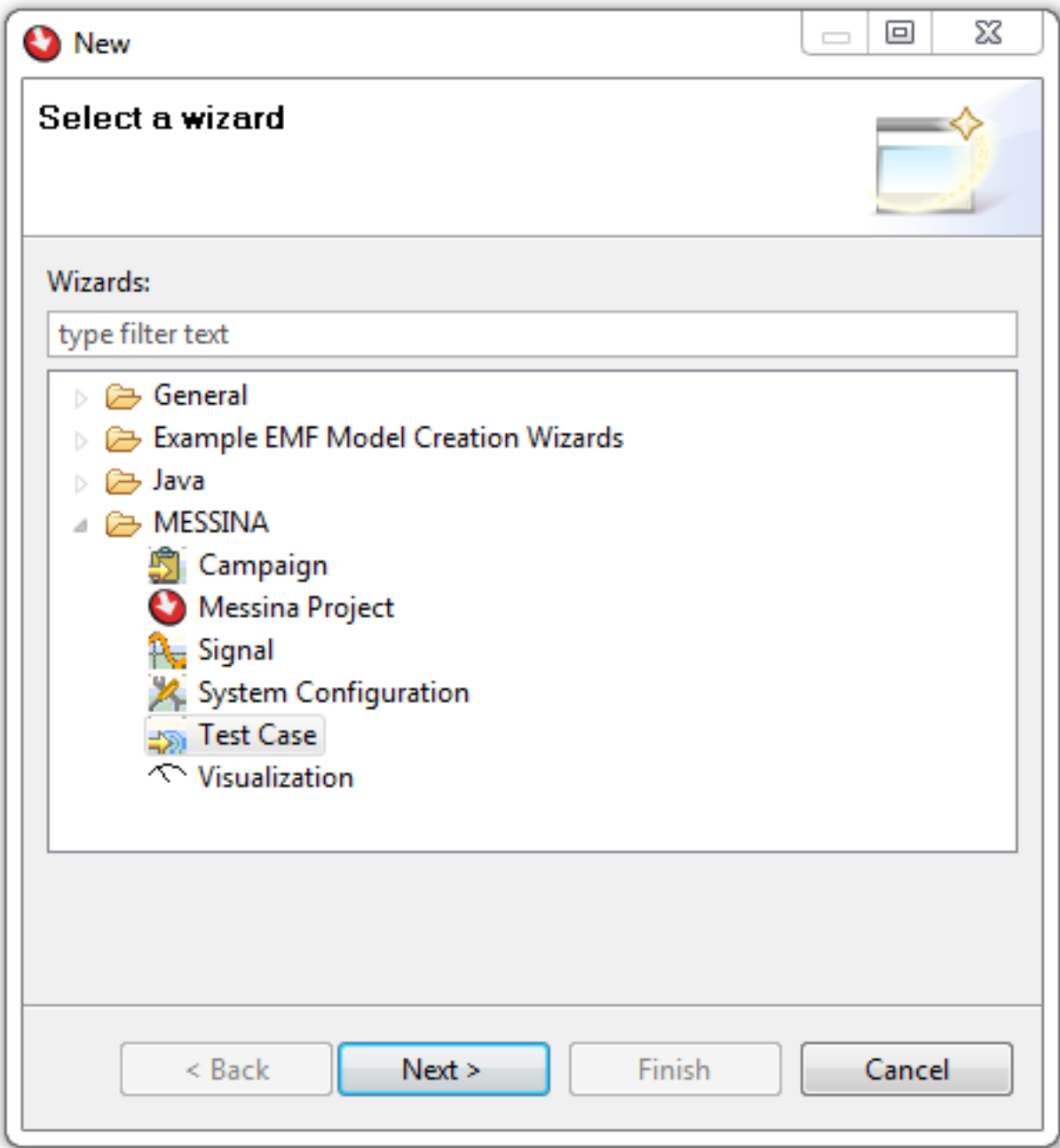
A **Test Case** must be defined and added to our project before we can execute a test. We must also add the source code to it. **Test cases** are listed under the **Test Cases** folder in the [Project Explorer](#). We will add a **Test Case** to our temperature compensation example which will be used to monitor and evaluate the compensated temperature. We will set a speed value and examine how the model reacts. The steps to do this are described in detail in the following sections.

### Add a Test Case

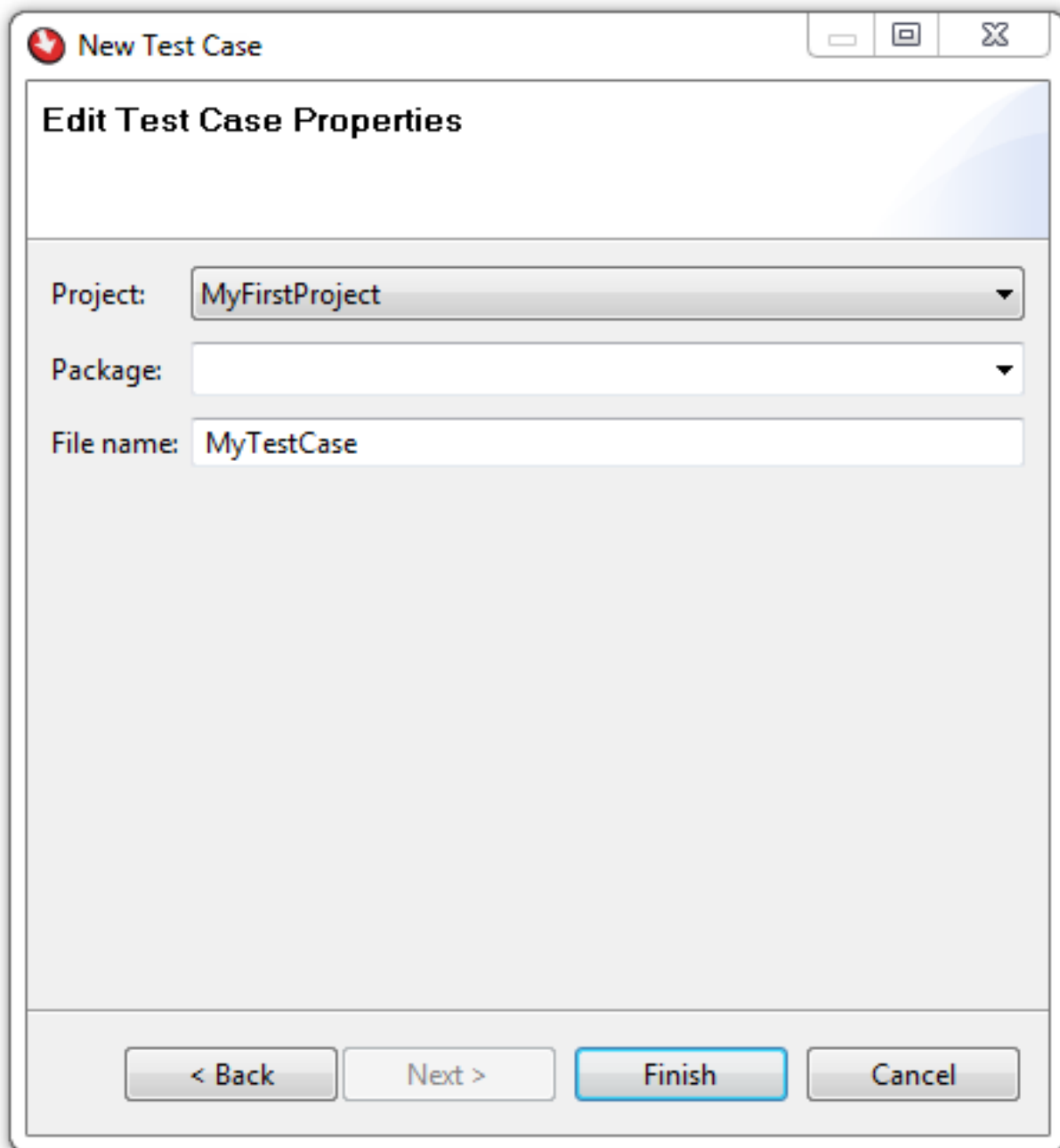
MESSINA **Test Cases** are created and edited using the **Designer** perspective which can be selected using the tabs at the top right of the main window. Switch to the **Designer** perspective.

From the main menu, open the dialog **File** → **New** → **Test Case**. It is also possible to open the **File** → **New** → **Other** dialog and select the **Test Case wizard** in the MESSINA folder.

Alternatively, the **Test Case Wizard** can be called using **Context Menu** → **New Wizards** → **Test Case**.



The following dialog box appears when the Test Case Wizard is called:



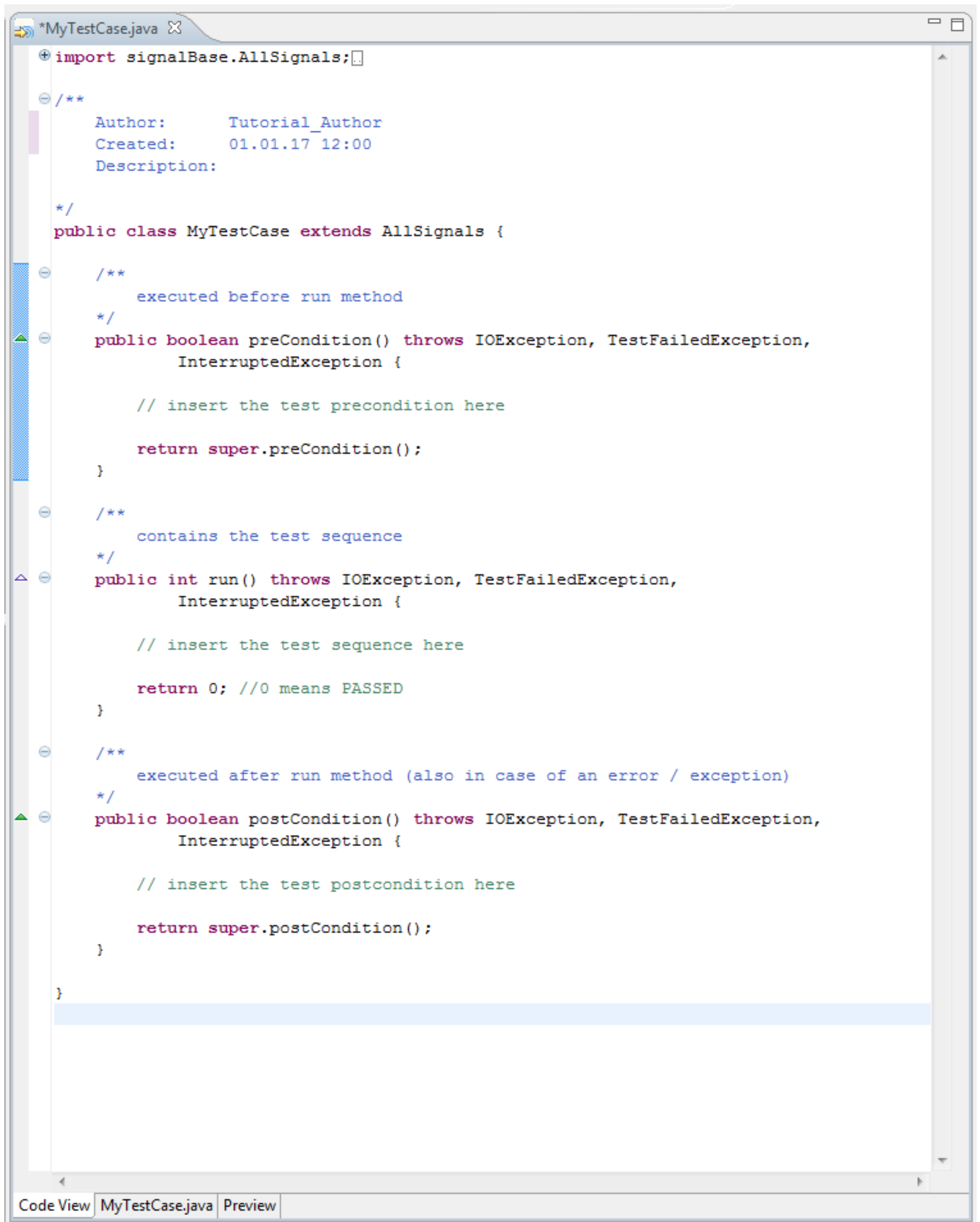
The **Project** pull-down list can be used to select the project where the **Test Case** is to be created. All projects listed in the [Project Explorer](#) will be available in the pull-down list. The currently active project is automatically set as the selected item. The **Package** text box can be used to group test cases. For our temperature compensation example packages will not be used and the text box shall remain empty.

The **File Name** text box is used to enter the **Test Case** name. This will be the name of the source code file which will be saved in the MESSINA project structure. A sample name is always generated automatically, but this can be changed to any **Test Case** name that does not already exist in the project. Enter **MyTestCase** as a test case name and press **Finish** to create the test case.

**Note:** Blank spaces are not allowed in the name.

When a new **Test Case** is created, the source file is created using the name entered above. The **Test Case** is automatically added to the Test Case folder in the [Project Explorer](#). The file is automatically assigned a file type ".java" to indicate that it is a java source file. A source code template consisting of an empty class is automatically generated as shown below.





```
import signalBase.AllSignals;

/**
 * Author: Tutorial_Author
 * Created: 01.01.17 12:00
 * Description:
 */
public class MyTestCase extends AllSignals {

    /**
     * executed before run method
     */
    public boolean preCondition() throws IOException, TestFailedException,
        InterruptedException {

        // insert the test precondition here

        return super.preCondition();
    }

    /**
     * contains the test sequence
     */
    public int run() throws IOException, TestFailedException,
        InterruptedException {

        // insert the test sequence here

        return 0; //0 means PASSED
    }

    /**
     * executed after run method (also in case of an error / exception)
     */
    public boolean postCondition() throws IOException, TestFailedException,
        InterruptedException {

        // insert the test postcondition here

        return super.postCondition();
    }
}
```

Now that a **Test Case** has been created, we can add the required source code to perform the actions

required. The `preCondition` will be used to initialize our **Test Case** before it is executed, and the `postCondition` to reset some values after execution is completed. Now we will [Program the Test Case](#)

## About the `preCondition` and `postCondition`

The **`preCondition`** function is executed before the `run` function. It can be used, for example, to perform an initialization or to make the required settings before a test can be executed. The **`postCondition`** function is executed after the **`run`** function. It can be used, for example, to reset items that may have been changed or to restore a default condition. The location of the **`preCondition`**, **`run`**, and **`postCondition`** functions within the **Test Case** class does not matter. Thus, the execution order within a **Test Case** will always be:

1. **`preCondition`** function
2. **`run`** function
3. **`postCondition`** function

## What do we want to do?

We will use the **TempComp** model and our **Test Case** to perform the following operations:

1. initialize the **`speed`** and **`sensor_temp`** signal values (done in the **`preCondition`** function)
2. wait for the **`air_temp`** signal to be equal to the **`sensor_temp`** signal value (done in the **`preCondition`** function)
3. set the **`speed`** signal directly in the source code to a fixed value (done in the **`run`** function)
4. wait for the **`air_temp`** signal to reach a value fixed in the source code (done in the **`run`** function)
5. reset the **`speed`** and **`sensor_temp`** values to zero (done in the **`postCondition`**)
6. wait for the **`air_temp`** signal to reach the **`sensor_temp`** signal value (done in the **`postCondition`**)

This might sound like a whole lot to do, but we will use built-in functions to handle each operation. This greatly simplifies programming **Test Cases** in MESSINA. Refer to the [Using MESSINA → Creating Tests → Signal Commands](#) section of this documentation for a complete description of built-in functions

## Programming the `preCondition` Function

The **Test Case** is programmed using the [Test Case Editor](#) in the Java language. It is not required that you be a Java expert to program a test case.

Signals available in the **Signalpool** are always available and can be referenced directly in the source code. The following example source code shows how to set the **`speed`** and **`sensor_temp`** to 0 and 10 respectively and how to wait for the **`air_temp`** signal to reach the **`sensor_temp`** signal value using the **`setValue`**, **`ASSERT`**, and **`waitValue`** built-in functions. This can be added to our **`preCondition`** function.

```
// initialize values (speed = 0, sensor_temp = 10)

speed.setValue(0);

sensor_temp.setValue(10);

// wait until air_temp == sensor_temp == 10

ASSERT(air_temp.waitValue(10, 60000), "Timeout-preCondition");
```

The last line will cause the process to wait for the **air\_temp** signal to be equal to the **sensor\_temp** signal. If this does not occur within the 60 second timeout, the "**timeout-preCondition**" message is returned. This should never happen because the **speed** = 0, and the model will adjust the **air\_temp** until it reaches the **sensor\_temp** value (no compensation is done when **speed** = 0). Refer to the **Using MESSINA** → **Creating Tests** → [Signal Commands](#) section of this documentation for a complete description of built-in functions.

## Programming the run Function

Our run function will be used to set the **speed** signal value to 130 and then wait for the **air\_temp** signal to reach 5. These values are given directly in the source code. Note that we have also added a timeout value of 30 seconds for the **waitValue** function.

```
// set speed value to 130

speed.setValue(130);

// wait for air_temp to reach 5

ASSERT(air_temp.waitValue(5, 30000), "Temperature not reached");
```

The last line performs a simple function. It waits for the **air\_temp** signal value to reach the value 5. If this does not occur within the **timeout** (set to 30 seconds), the "**Temperature not reached**" message is returned. Refer to the **Using MESSINA** → **Creating Tests** → [Signal Commands](#) section of this documentation for a complete description of built-in functions.

## Programming the postCondition Function

Add the following source code to the ***postCondition*** function:

```
// reset speed and sensor_temp signals

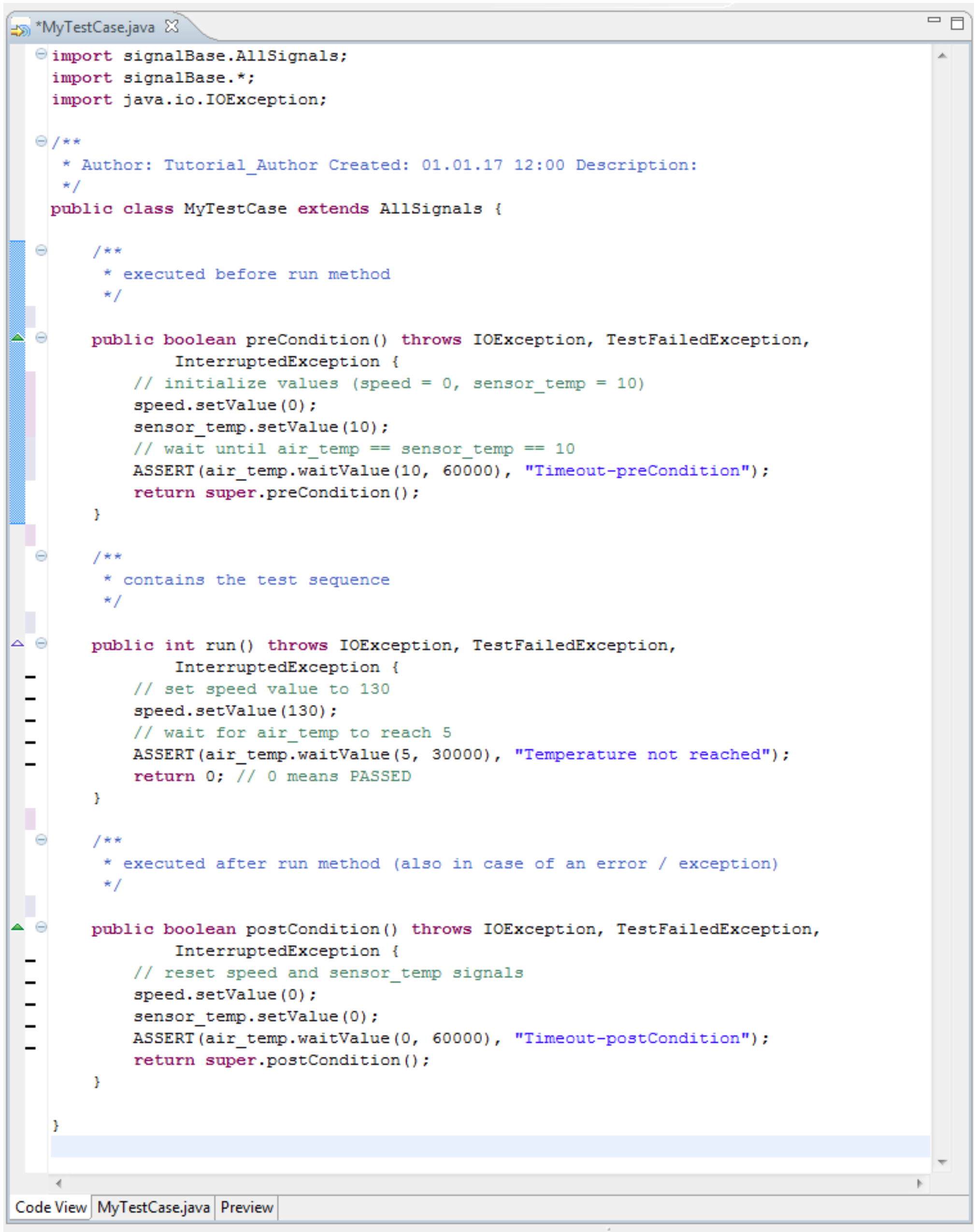
speed.setValue(0);

sensor_temp.setValue(0);

ASSERT(air_temp.waitValue(0, 60000), "Timeout-postCondition");
```

The signals are reset to 0, and we wait until ***air\_temp*** and ***sensor\_temp*** equal 0. If this does not occur within the 60 second timeout, the "***timeout-postCondition***" message is returned.

The programming of our ***Test Case*** is now completed! Here is a picture of the completed ***Test Case***:



```
*MyTestCase.java
import signalBase.AllSignals;
import signalBase.*;
import java.io.IOException;

/**
 * Author: Tutorial_Author Created: 01.01.17 12:00 Description:
 */
public class MyTestCase extends AllSignals {

    /**
     * executed before run method
     */

    public boolean preCondition() throws IOException, TestFailedException,
        InterruptedException {
        // initialize values (speed = 0, sensor_temp = 10)
        speed.setValue(0);
        sensor_temp.setValue(10);
        // wait until air_temp == sensor_temp == 10
        ASSERT(air_temp.waitValue(10, 60000), "Timeout-preCondition");
        return super.preCondition();
    }

    /**
     * contains the test sequence
     */

    public int run() throws IOException, TestFailedException,
        InterruptedException {
        // set speed value to 130
        speed.setValue(130);
        // wait for air_temp to reach 5
        ASSERT(air_temp.waitValue(5, 30000), "Temperature not reached");
        return 0; // 0 means PASSED
    }

    /**
     * executed after run method (also in case of an error / exception)
     */

    public boolean postCondition() throws IOException, TestFailedException,
        InterruptedException {
        // reset speed and sensor_temp signals
        speed.setValue(0);
        sensor_temp.setValue(0);
        ASSERT(air_temp.waitValue(0, 60000), "Timeout-postCondition");
        return super.postCondition();
    }

}
```

Code View MyTestCase.java Preview


We are now ready to execute the ***Test Case***. This will be done in the next section.

## Step 9 - Execute the Test Case

In this section we will execute the previously created **Test Case**. We will use the [Table View visualization](#) to examine the process while it is running. This will be done in the **Executor** perspective with the **Table View visualization** selected and connected.

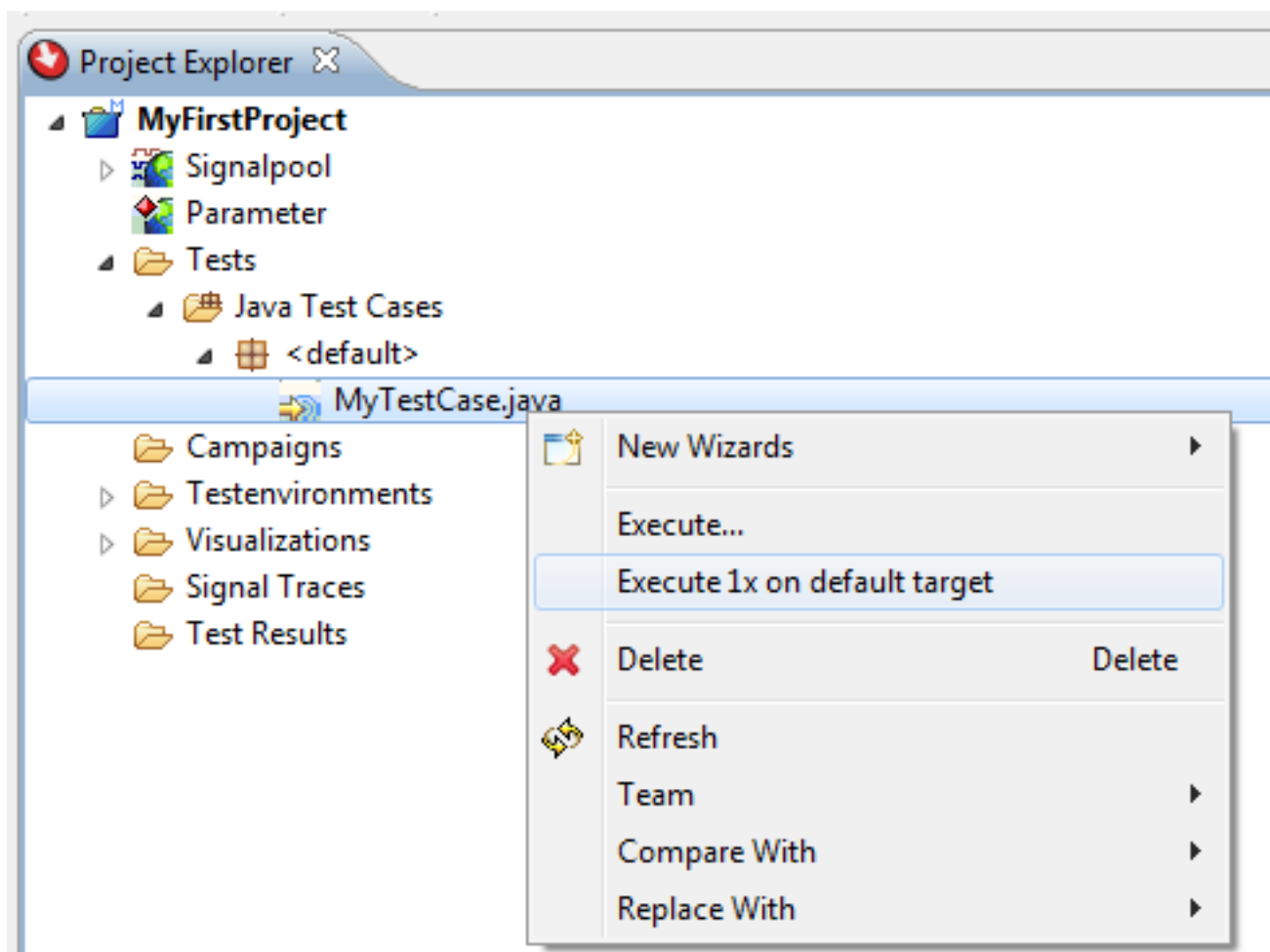
### Execution of a Test Case - Table View

Make sure that the **Table View visualization** is being displayed and **connected**. You can **connect**

the visualization by pressing the  icon at the top of the **Table View visualizations** view. Make sure that the SiL target shown in the **Target Manager** is set as the default target. This can be done by selecting the target, then calling the **Context Menu** → **Set as Default Target**. A small overlay icon

with a check mark  indicates the currently selected default target.

Open the **Test Case** folder in the [Project Explorer](#), mark the **Test Case** we created earlier (**MyTestCase** in our example) and call the context menu. Select the **Execute 1x on default target** option as shown in the following picture:



Watch the values in the [Table View visualization](#) and convince yourself that they react correctly. The next section will perform the same operation but we will use the [Control Panel](#) we previously created and its graph element to view the results. The process will be easier to following using a graph.

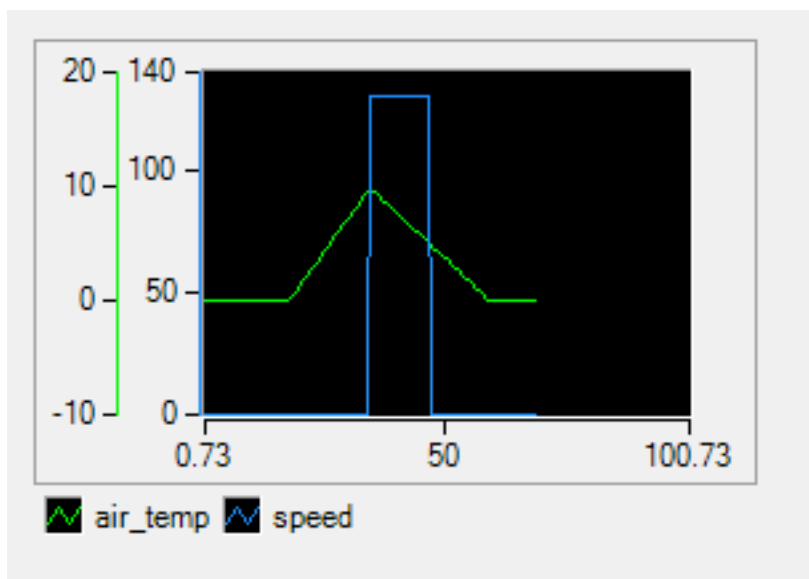


## Execution of a Test Case - Control Panel Graph

Select the [Control Panel visualization](#) by clicking on the correct tab in the visualization view and make sure that it is **connected** by pressing the icon.

Open the **Test Case** folder in the [Project Explorer](#), mark the **Test Case** we created earlier (**MyTestCase** in our example) and call the context menu. Select the **Execute 1x on default target** option as we did in the last section.

Watch the values in the graph and convince yourself that they react correctly. The graph should look something like this:

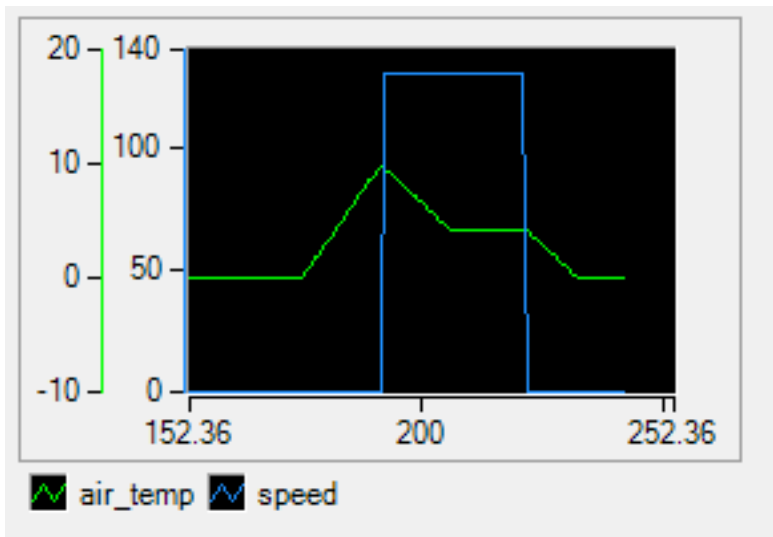


### What do we see?

The plot is flat before we start execution (**speed** = 0, **air\_temp** = 0). Once the execution starts, the **air\_temp** signal begins to move the value of the **sensor\_temp** signal (which we set to 10). This is the **preCondition** function being executed. Once the condition **air\_temp** = **sensor\_temp** is fulfilled, the **preCondition** function is complete and the **run** function is executed. The **speed** jumps to 130. Because of the increase in **speed**, the model begins to compensate, this reduces the **air\_temp** value. Once the **air\_temp** value reaches the value we set (**air\_temp** = 5), the **run** function is completed and the **postCondition** function is called which sets the **speed** and **sensor\_temp** to 0. The model now slowly moves the **air\_temp** value to 0. Once there, the execution of our **Test Case** is completed.

Now, let's adjust the **air\_temp** to a lower value. Modify the **air\_temp** value in the **Test Case** so that the value is 1 (previously it was 5). Execute the **MyTestCase Test Case** again. A graph that looks something like this will be the result:





The plot is flat before we start execution as expected. Once the execution starts, the ***air\_temp*** signal begins to move the value of the ***sensor\_temp*** signal (***preCondition*** function). Once the condition ***air\_temp = sensor\_temp*** is fulfilled, the ***preCondition*** function is complete and the ***run*** function is executed. So far, so good. The speed ***jumps*** to 130. The increase in ***speed*** causes the model to compensate the ***air\_temp*** value as happened before. After a short time, the ***air\_temp*** signal stays constant. This indicates that the model has fully compensated the ***air\_temp*** for the ***speed*** value. Remember we set the ***air\_temp*** to 1, but the model stopped compensating at 4.2 (OK, I peeked at the [Table View](#) to see the exact value). In other words, our ***air\_temp*** value will never be reached. This is why we have a ***timeout***. Once the ***timeout*** is reached, the ***run*** function execution is ended, and the ***postCondition*** is executed. We can see this because the ***speed*** goes to 0. The model then compensates the ***air\_temp*** value back to 0 (remember we set the ***sensor\_temp*** value to 0 in the ***postCondition*** function) before execution is completed.

## Step 10 - Create a Campaign and Add Parameters

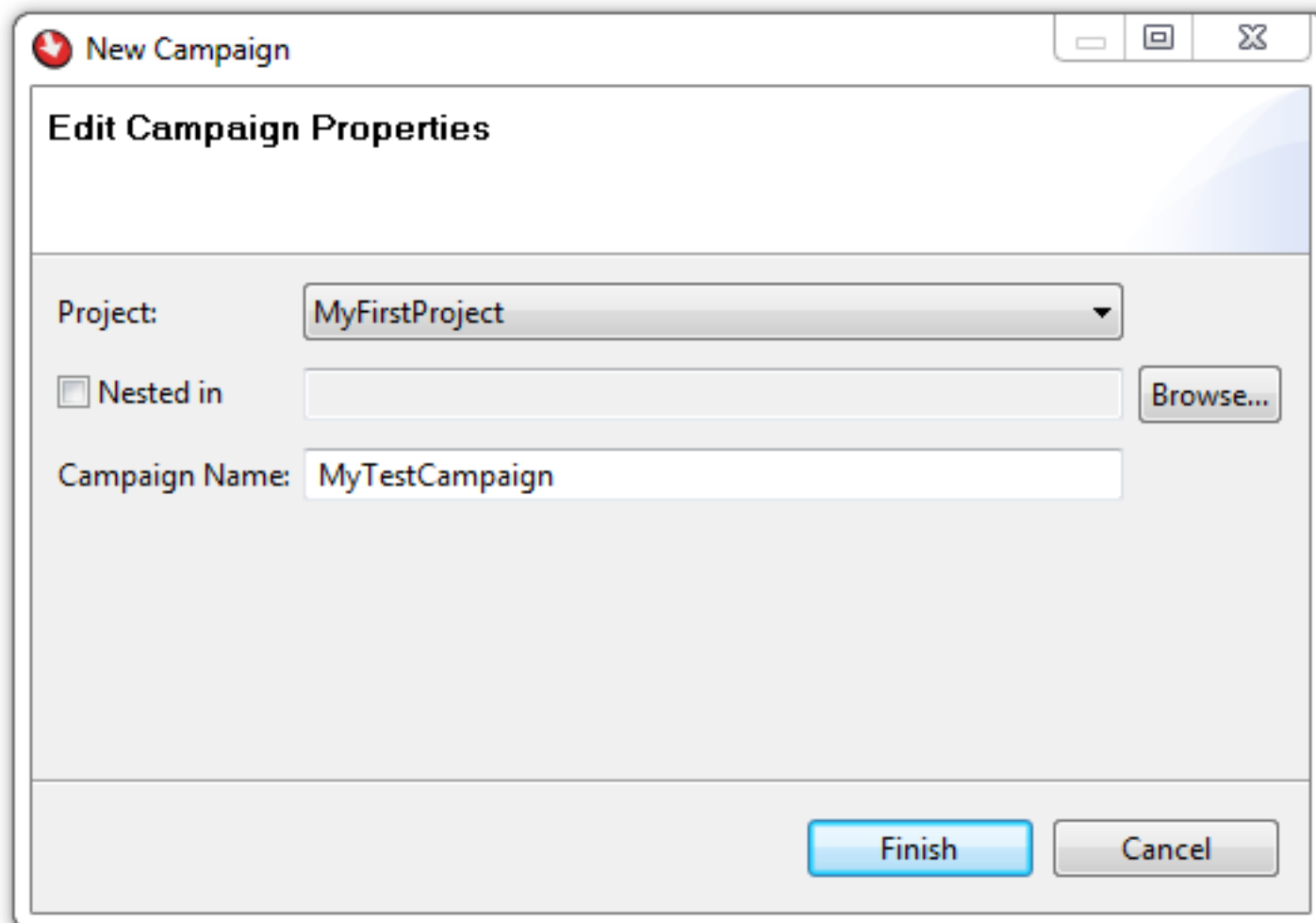
Now that we have a **Test Case** we can add this to a **Campaign** which we will create and add to the project. We will add the previously created **Test Case** to a **Campaign** to our temperature compensation example which will be used to monitor and evaluate the compensated temperature. We will now use **Parameters** to set a **speed** value instead of setting this directly as we did before. The steps to do this are described in detail in the following sections.

### Add a Campaign

A MESSINA **Campaign** is defined as a series of **Test Cases** performed in the order they are listed in the [Project Explorer](#). **Campaigns** are a flexible way of designing a complete test system. We will use our temperature compensation example to demonstrate the creation and execution of a simple **Campaign**. We will also use **Parameters** instead of fixed values in the code to test the model.

MESSINA **Campaigns** are also created and edited using the **Designer** perspective. From the main menu, open the dialog **File** → **New** → **Campaign**. It is also possible to open the **File** → **New** → **Other** dialog and select the **Campaign wizard** in the MESSINA folder.

Alternatively, the **Test Case Wizard** can be called using the **Context Menu** → **New Wizards** → **Campaign**.



The **Project** pull-down list can be used to select the project where the **Campaign** is to be created. All

projects listed in the [Project Explorer](#) will be available in the pull-down list. The currently active project is automatically set as the selected item. It is possible to "nest" campaigns (have a campaign within another campaign). If this is required, the **Nested In** box must be checked, and the location where the new **Campaign** is to be created must be selected by hand. For our temperature compensation example there will be no nested **Campaigns**.

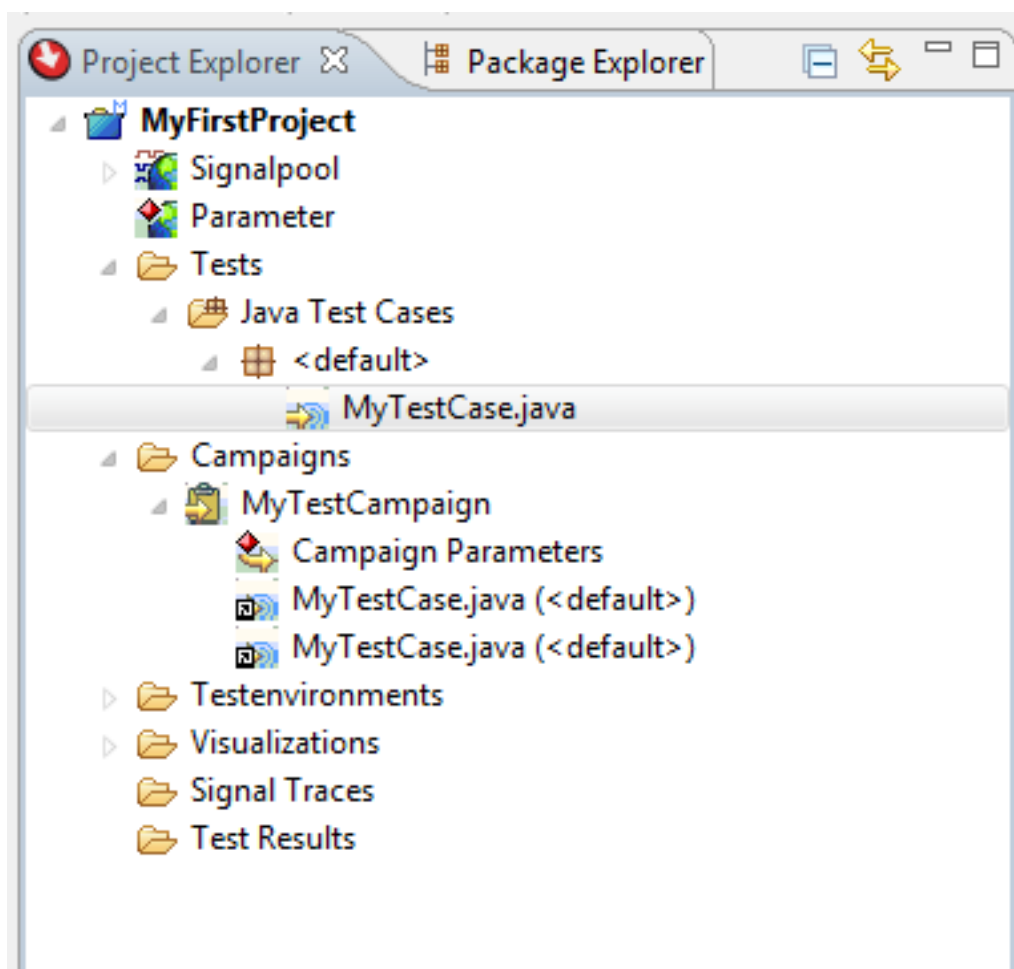
The **Campaign Name** text box is used to enter the campaign name. This will be the name displayed in the **Project Explorer**. A sample name is always generated automatically, but this can be changed to any **Campaign** name that does not already exist in the project. Enter **MyTestCampaign** as the campaign name and press **Finish** to create the **Campaign**.

**Note:** Blank spaces are not allowed in the name.

## Adding Test Cases to the Campaign

Once the **Campaign** is created, the **Test Cases** can be added to the **Campaign** by marking the **Test Case** and using drag-and-drop to add it to the **Campaign** item directly in the **Project Explorer**. Add the same **Test Case** to the **Campaign** twice. We will use the 2 **Test Cases** with different parameters to create different tests within our **Campaign**.

The MESSINA **Project Explorer** view now shows a **Campaign** containing two versions of the same **Test case** and should look like this:



## Adding Project Parameters

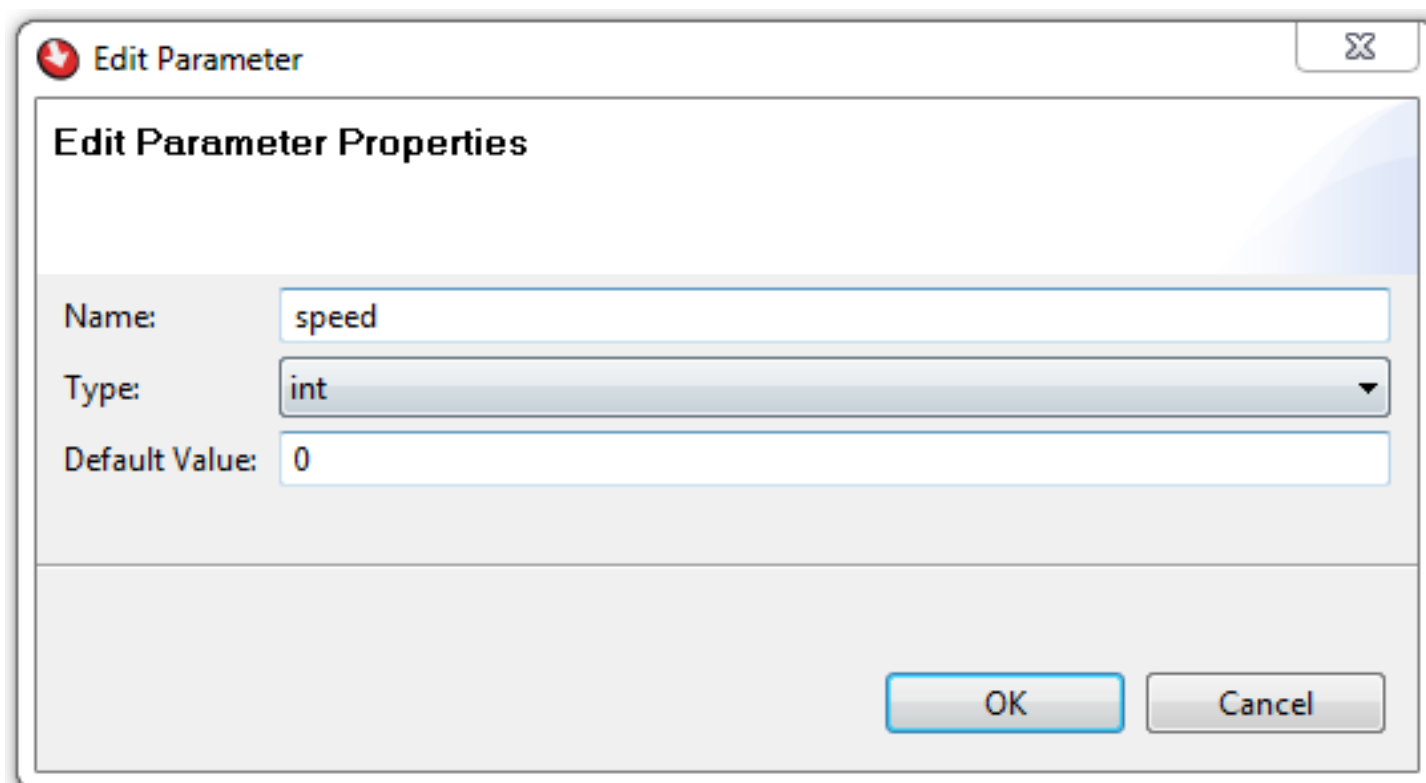
Now we will take a closer look at how **Parameters** can be used in MESSINA. This will be done from the **Designer** perspective. We will add several parameters to our **TempComp** example to show how they

are used and the flexibility that they offer compared to using fixed values as we did in the previous step.

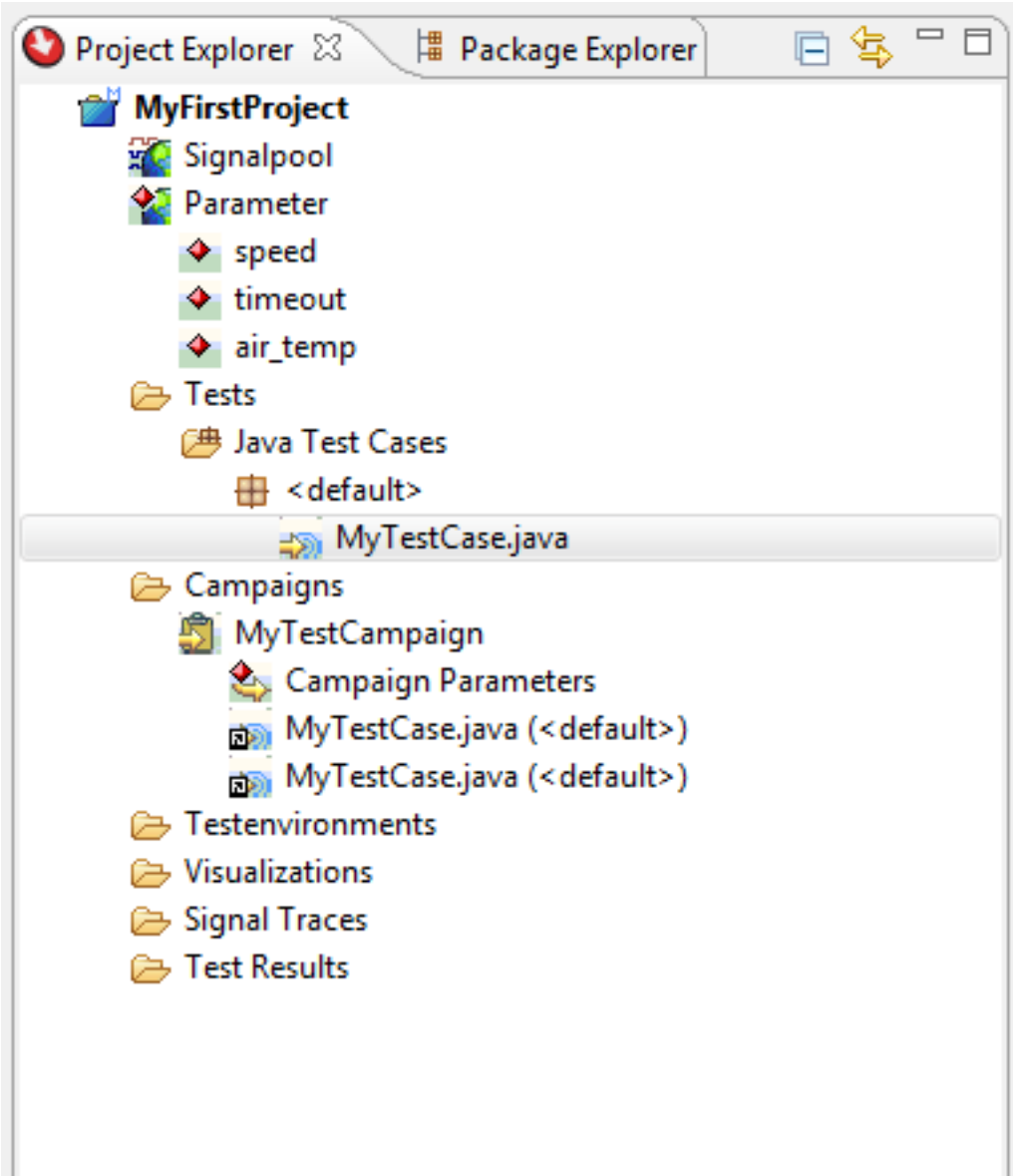
**Parameters** in the MESSINA development environment are like variables used within the MESSINA project. **Parameters** can be set and used at different points in MESSINA. For example, you can add **Parameters** in the [Project Explorer](#) view under the **Parameter** item (see below for details on how to add parameters). This can be used, for example, to set **Parameters** to default values within a MESSINA project. **Parameters** can also be used to set values for use only within a **Campaign**. It is also possible to assign **Parameters** for use only within a given **Test Case**. This gives you a lot of flexibility by enabling different tests to be performed using the same **Test Case** with different parameter values.

The sections below will show how to add **Parameters** used to set default values. We will then create a **Campaign** and add **Parameters** to the **Campaign**. We will add **Parameters** called **timeout**, **air\_temp**, and **speed**. These will be used to set signals within our **Test Case** (which will be added in the next section), to compare a signal level, and to set a timeout for our **Test Case**.

In the **Project Explorer** window, add a parameter called **speed** by selecting the **Parameter** item and calling **Context Menu** → **New Parameter**. The following dialog is called:

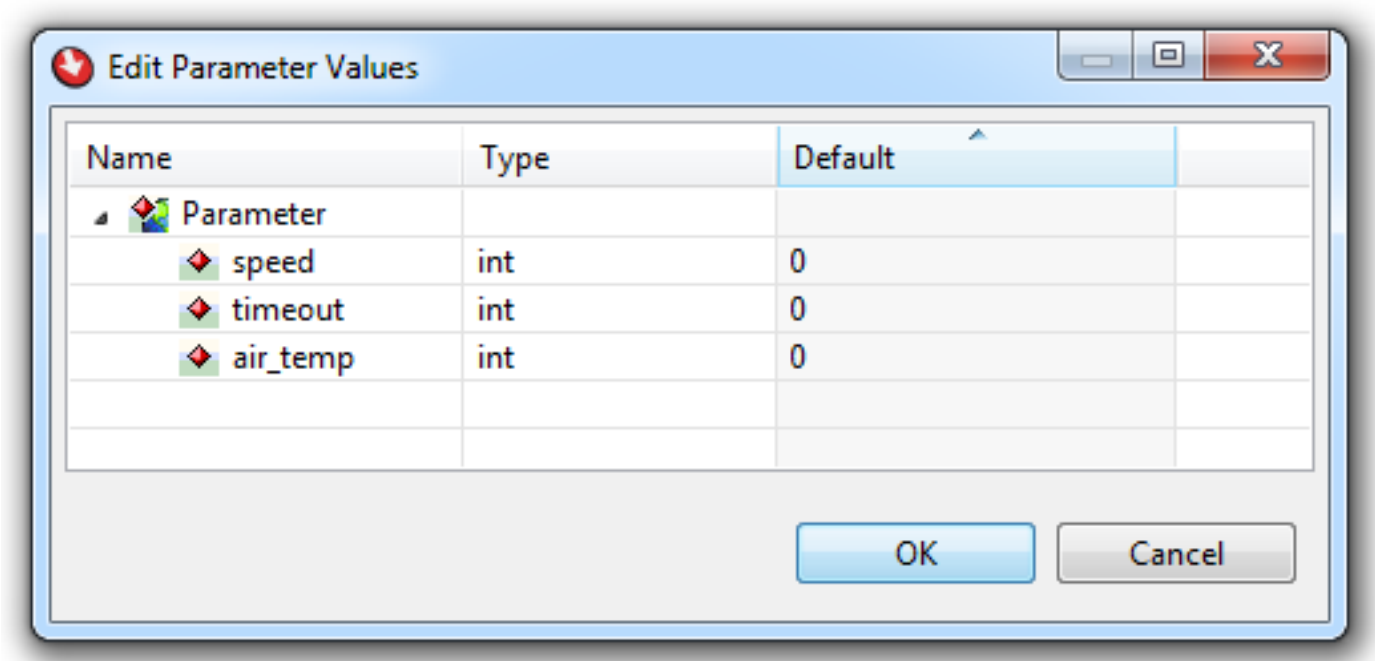


Enter the parameter **Name** as shown (**speed** in this case), the **Type** and **Default Value** can remain as displayed. After pressing **OK**, the **Parameter** is added to the [Project Explorer](#) (expand the **Parameter** item) and to the [Parameter Manager](#) window. Add **Parameters** for **timeout** and **air\_temp** the same way. The **Project Explorer** should now look like this:



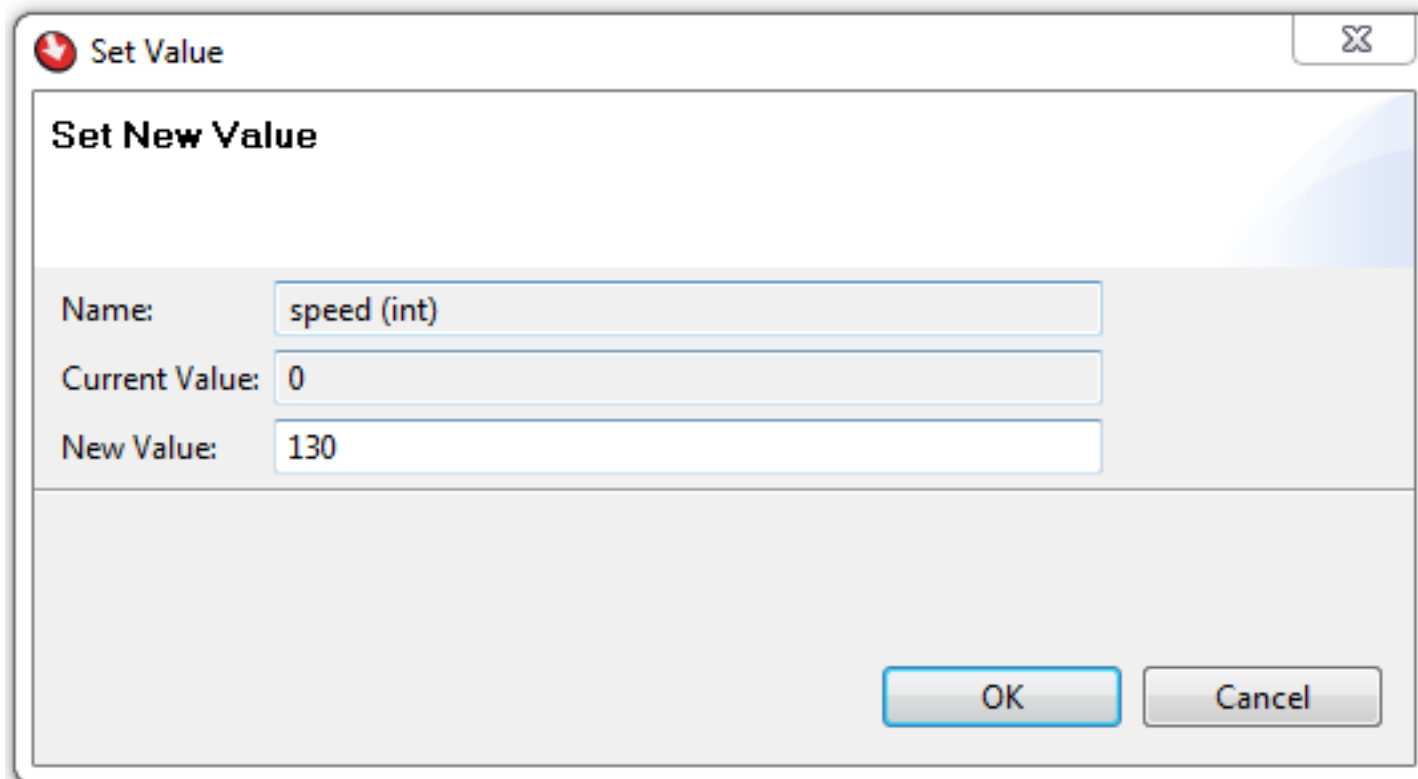
## Setting Parameters for a Campaign

When a new **Campaign** is created (as we did in the previous section) a **Campaign Parameters** item is automatically created under that **Campaign** name. Initially it will be empty. Double click on the **Campaign Parameters** item, and the following dialog appears:



Notice that the **Type** and **Default** value of the **Parameter** are shown. The **Default** value is the value we set when we created the **Parameter** for the project in the previous section.

Now, double click on the **speed** parameter in the list. The following dialog appears:



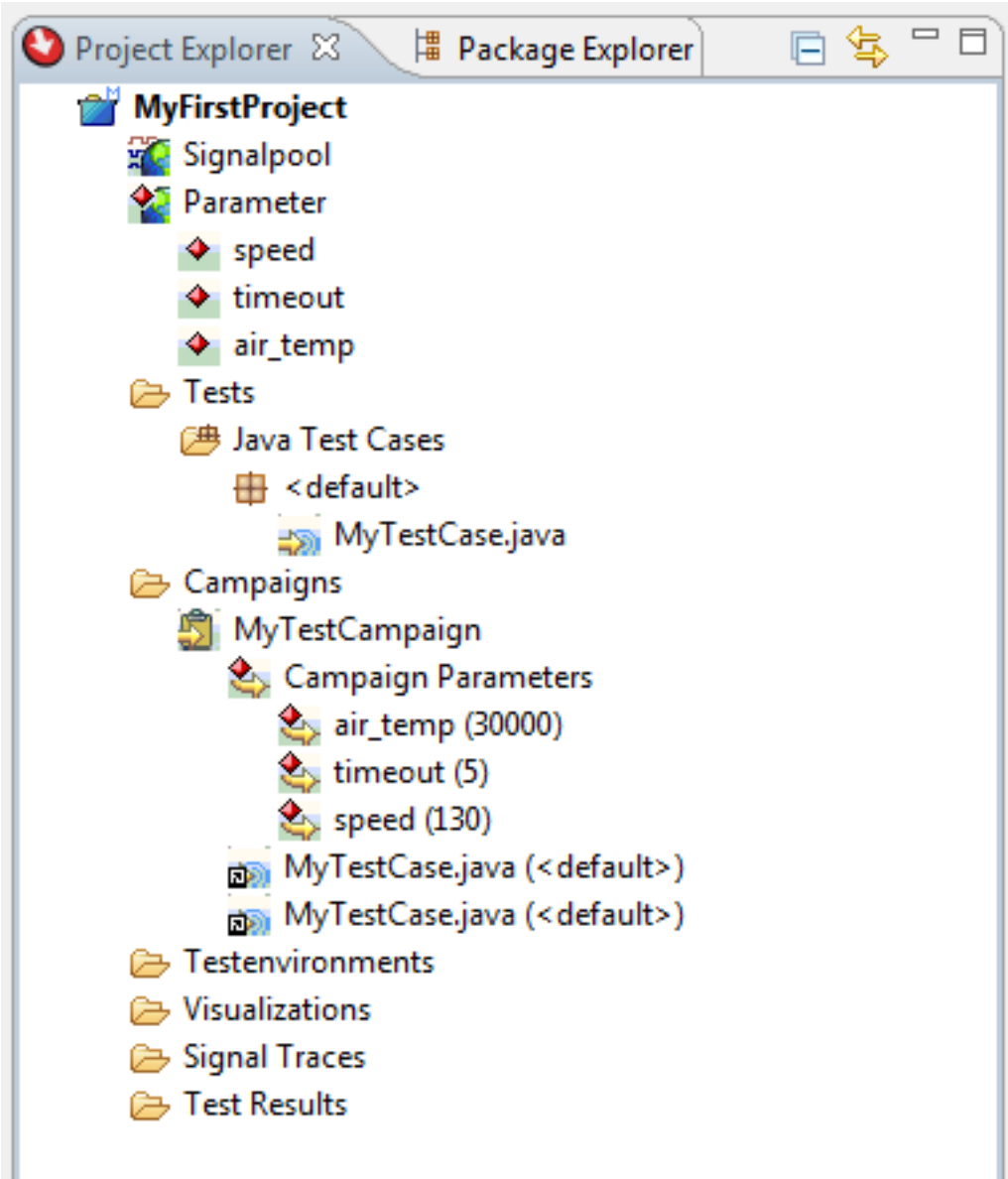
This dialog allows us to set a **New Value** for the selected **Parameter**. The value set here will only be valid within the **Campaign** where it is listed. For our temperature compensation example we will require a different value for the **speed Parameter**. Set the **New Value** of the **speed Parameter** to 130.

Use the same procedure to add the **timeout** and **air\_temp Parameters** to the **Campaign**. Make the following settings:

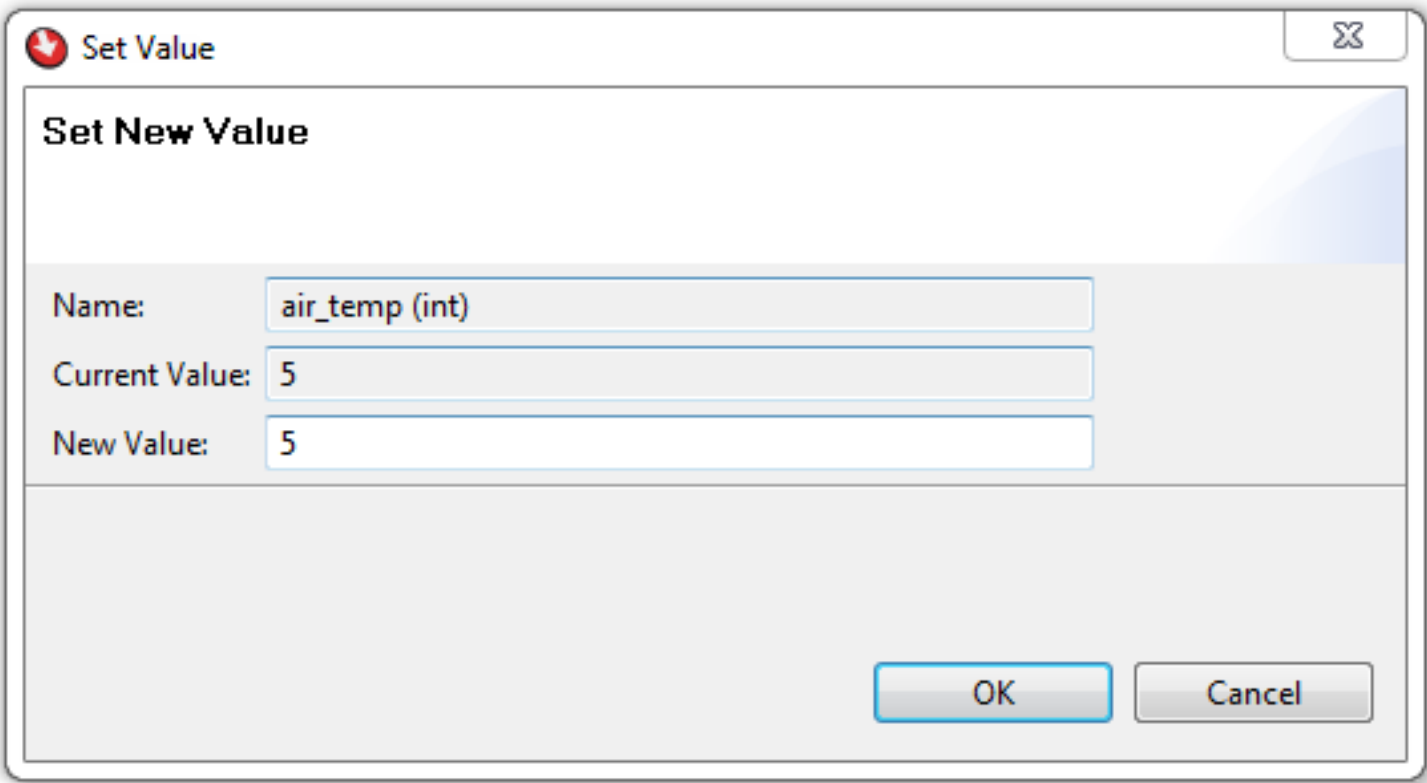
- Set **timeout** = 30000 (a value in milli-seconds)
- Set **air\_temp** = 5
- Set **speed** = 130

We will use these settings within the first **Test Case** to perform our tests.

The [Project Explorer](#) view should have the **project parameters** and **Campaign parameters**. If you expand the items, it should now look like this:



To change a **Parameter** value within the **Campaign**, double click on the desired parameter and change the value directly in the dialog box as shown below.



To remove a parameter from the **Campaign**, use the **Context Menu** → **Un-override** option. This removes the **Parameter** from the **Campaign** only, the project **Parameter** remains unchanged.

## Modify the Test Case to use Parameters



We will use the **TempComp** model and our **Test Cases** to perform the following operations:

1. initialize the **speed** and **sensor\_temp** signal values (same as before)
2. wait for the **air\_temp** signal to be equal to the **sensor\_temp** signal value (same as before)
3. set the **speed** signal to our **speed parameter** value (done in the **run** function)
4. wait for the **air\_temp** signal to reach the **air\_temp parameter** value (done in the **run** function)
5. reset the **speed** and **sensor\_temp** values to zero (same as before)
6. wait for the **air\_temp** signal to reach the **sensor\_temp** signal value (same as before)

Again, we will use built-in functions to handle each operation.

## Programming the preCondition Function

The **preCondition** function remains the same as before.

```
// initialize values (speed = 0, sensor_temp = 10)

speed.setValue(0);

sensor_temp.setValue(10);

// wait until air_temp == sensor_temp == 10

ASSERT(air_temp.waitValue(10, 60000), "Timeout-preCondition");
```

## Modifying the run Function to use Parameters

**Parameters** are referenced by the "**params.**" prefix. By typing "**params.**" you can see a listing of all parameters available. Use the arrow keys to move to the desired parameter.

**Note:** the **Parameter** values used will be those set within the **Campaign** (not the project default value settings).

```
// set speed value to parameter

speed.setValue(params.speed);

// wait for air_temp to reach the parameter value

ASSERT(air_temp.waitValue(params.air_temp, params.timeout), "Temperature not reached");
```



The fixed values we used before have now been replaced with the ***Parameters*** we created. These are highlighted in the source code above for clarity. The operation of the run function remains unchanged.

## Programming the postCondition Function

The ***postCondition*** function remains the same as before.

```
// reset speed and sensor_temp signals

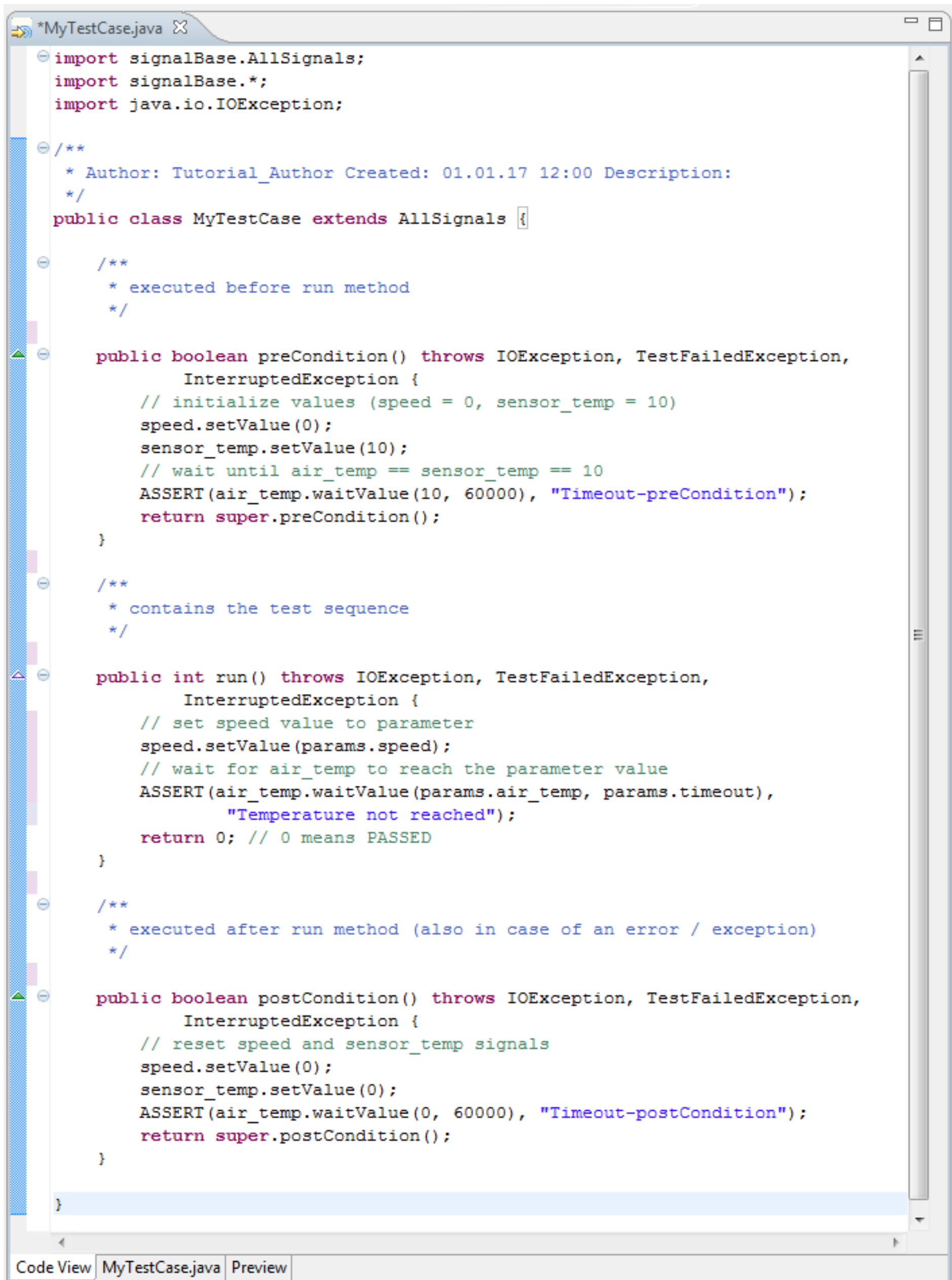
speed.setValue(0);

sensor_temp.setValue(0);

ASSERT(air_temp.waitForValue(0, 60000), "Timeout-postCondition");
```

The signals are reset to 0, and we wait until ***air\_temp == sensor\_temp == 0***. If this does not occur within the 60 second timeout, the "***timeout-postCondition***" message is returned.

The modified source code of our ***Test Case*** is now completed! Here is a picture of the completed ***Test Case***:



```
*MyTestCase.java
import signalBase.AllSignals;
import signalBase.*;
import java.io.IOException;

/**
 * Author: Tutorial_Author Created: 01.01.17 12:00 Description:
 */
public class MyTestCase extends AllSignals {

    /**
     * executed before run method
     */

    public boolean preCondition() throws IOException, TestFailedException,
        InterruptedException {
        // initialize values (speed = 0, sensor_temp = 10)
        speed.setValue(0);
        sensor_temp.setValue(10);
        // wait until air_temp == sensor_temp == 10
        ASSERT(air_temp.waitValue(10, 60000), "Timeout-preCondition");
        return super.preCondition();
    }

    /**
     * contains the test sequence
     */

    public int run() throws IOException, TestFailedException,
        InterruptedException {
        // set speed value to parameter
        speed.setValue(params.speed);
        // wait for air_temp to reach the parameter value
        ASSERT(air_temp.waitValue(params.air_temp, params.timeout),
            "Temperature not reached");
        return 0; // 0 means PASSED
    }

    /**
     * executed after run method (also in case of an error / exception)
     */

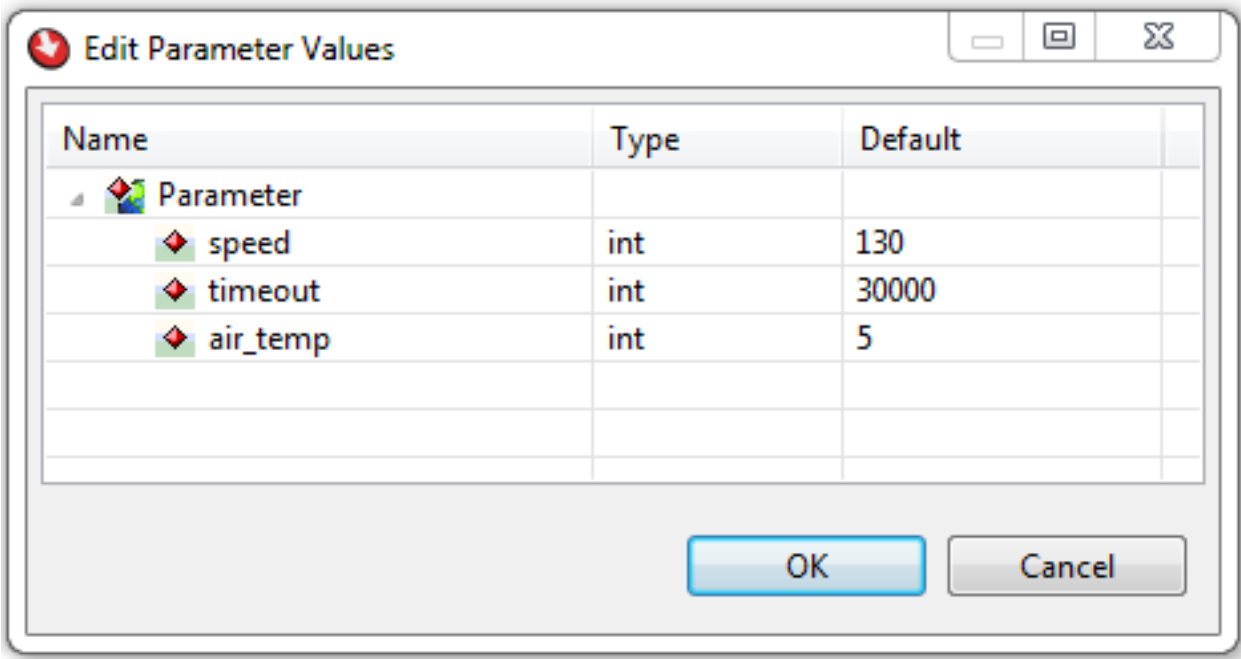
    public boolean postCondition() throws IOException, TestFailedException,
        InterruptedException {
        // reset speed and sensor_temp signals
        speed.setValue(0);
        sensor_temp.setValue(0);
        ASSERT(air_temp.waitValue(0, 60000), "Timeout-postCondition");
        return super.postCondition();
    }
}
```

Code View MyTestCase.java Preview

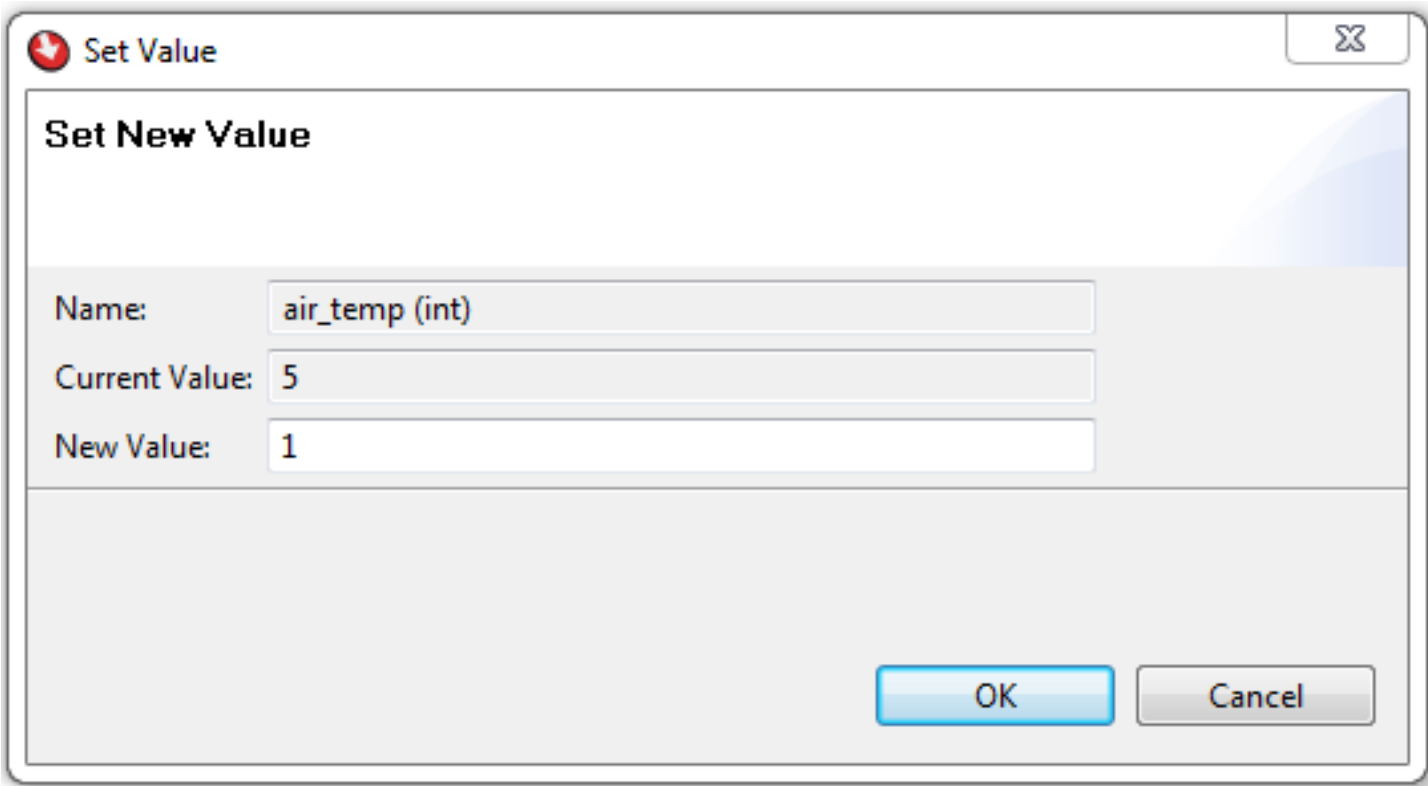
## Setting Parameters for Each Test Case

**Parameters** can be set differently not only for each **Campaign**, but also for each **Test Case**. Now the real power of **Parameters** comes into play. We will use this feature to create two different tests based on the same **Test Case**, but using different parameters.

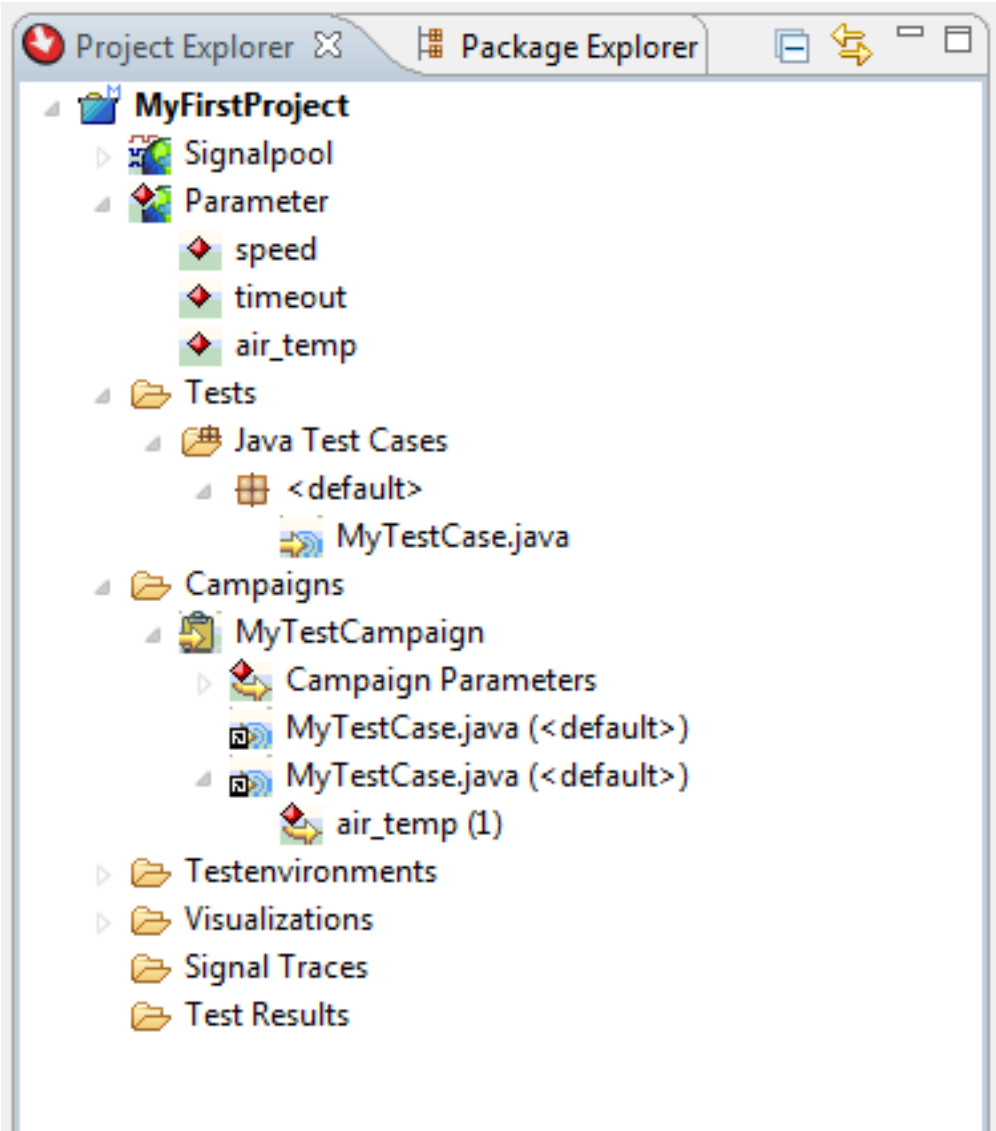
Double click on the first instance of the **MyTestCase Test Case** we created. The following dialog appears:



These are the values we set for our **Campaign**. They are currently relevant for both the first and second instance of the **Test Case**. Press **OK**, we do not need to change the parameters for the first **Test Case**. Now double click on the second instance of **MyTestCase** within the **Campaign**. The same dialog with the same values appears. Double click on the **air\_temp** entry. The following dialog appears:



Set the **New Value** to 1 as shown above and press **OK**. The [Project Explorer](#) view should now look like this:



Now we have the parameters we created for our project with the assigned default values. We also assigned different values to them for our **MyTestCampaign Campaign**. After that, we assigned a different value to the **air\_temp parameter** for the second instance of our **MyTestCase Test Case**. This is summarized in the table below:

Parameter Name	Value (Project/default)	Value - MyTestCampaign	Value - MyTestCase (1)	Value - MyTestCase(2)
<b>speed</b>	0	130	130	130
<b>timeout</b>	0	30000	30000	30000
<b>air_temp</b>	0	5	5	1

If a parameter is not assigned a new value for a **Test Case**, the value for the **Campaign** is used. If a parameter is not assigned a new value for a **Campaign**, then the project parameter (default) value is used. In our example, only the **air\_temp** parameter was assigned a new value for the second instance of our **Test Case** within the **Campaign**. To remove a parameter assignment, select the desired item and use **Context Menu → Un-override**.

We are now ready to execute the **Campaign**.

## Step 11 - Execute the Campaign

In this section we will execute the previously created [Test Campaign](#). We will use the [Table View](#) and [Control Panel visualizations](#) to examine the process. This will be done in the **Executor** perspective with the **Table View visualization** selected and connected.

### What do we have?

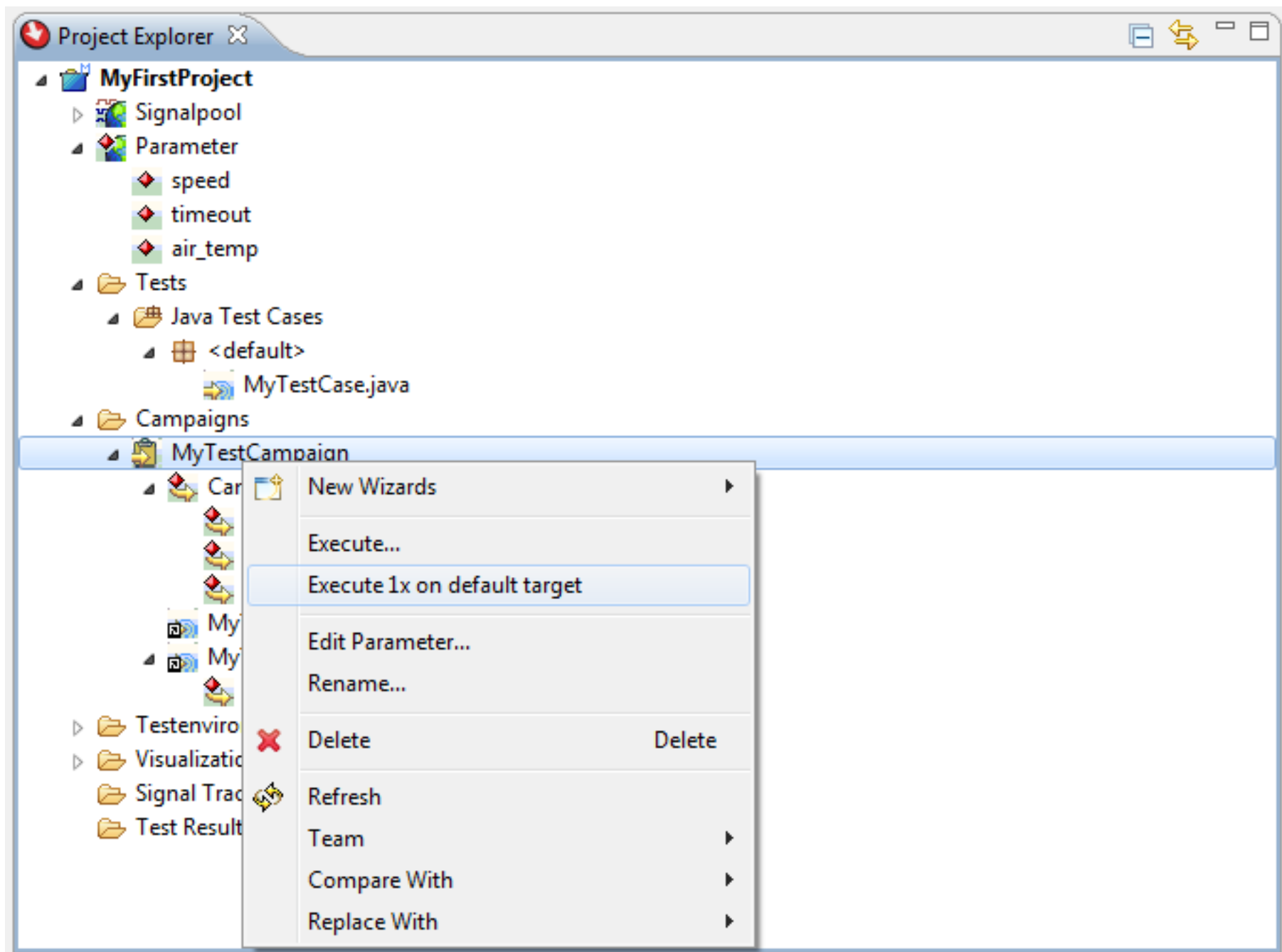
We want to execute our **Campaign** with the settings and parameters added in the last step. We have a **Campaign** with 2 **Test Cases**. The process (source code) is the same for each **Test Case**, but we intentionally set a **Parameter** to a different value in the second **Test Case**. In other words, we want to test different conditions without having to create or modify a **Test Case**.

### Execution of a Campaign - Table View

Make sure that the **Table View visualization** is being displayed and that it is **connected**. You can

**connect** the visualization by pressing the  icon at the top of the **Table View** visualization view.

Open the Campaign folder in the [Project Explorer](#), mark the **Campaign** we created earlier (**MyTestCampaign** in our example) and call the context menu. Select the **Execute 1x on default target** option as shown in the following picture:



Remember that we are now executing 2 separate **Test Cases** and that the second **Test Case** uses a different parameter for the **air\_temp**. The first **Test Case** will execute as before. The second **Test Case** uses an **air\_temp** parameter value that was selected so that the **TempComp** model will stop compensating the value before it is reached. This causes the timeout condition we set in the **run** function.

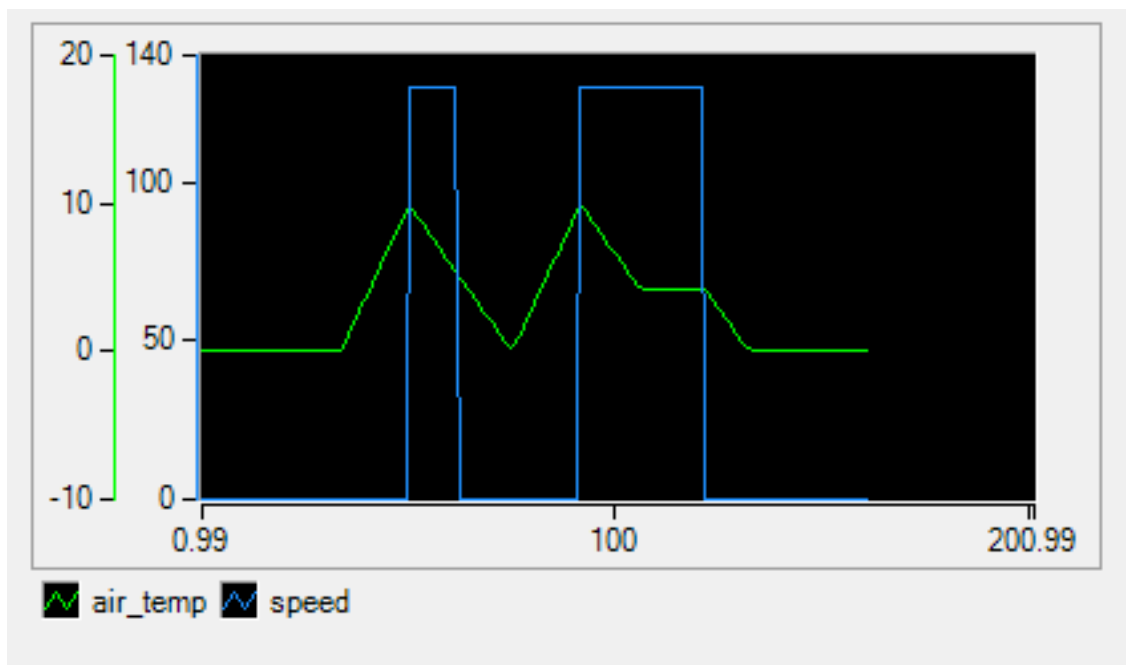
Watch the values in the [Table View visualization](#) and convince yourself that they react correctly. It is much easier to examine the results using a graph so let's use the [Control Panel](#) visualization we created earlier to do this again.

## Execution of a Campaign - Control Panel Graph

Select the **Control Panel visualization** by clicking on the correct tab in the visualization view and make sure that it is **connected**.

Open the Campaign folder in the [Project Explorer](#), mark the **Campaign** we created earlier (**MyTestCampaign** in our example) and call the context menu. Select the **Execute 1x on default target** option as we did in the last section.

Watch the values in the graph and convince yourself that they react correctly. The graph should look something like this:



## What do we see?

The same as before, but this time all in one process. The plot is flat before execution begins, then the **air\_temp** signal begins to move to the **sensor\_temp** value. Once they are equal, the **preCondition** function of the first **Test Case** is completed. The **speed** then jumps to the parameter value, the model begins to compensate (reducing the **air\_temp** value) as before. Once the **air\_temp** parameter value is reached, the **run** function is completed and the **postCondition** function is executed. The model compensates the **air\_temp** value to 0. Now, the second **Test Case** begins. The **preCondition** function is the same, so the **air\_temp** signal begins to move to the **sensor\_temp** value. Once equal, the **speed** jumps to the parameter value and the model compensation takes place. At some point, the model stops compensating, but the **air\_temp** parameter value is not reached. Eventually, the **timeout** condition occurs, the **speed** goes to 0, and the model compensates the **air\_temp** down to 0.

You might have noticed that something happened in the [Result Manager](#) view after each execution. We will examine this more closely in the next section.



## Step 12 - The Result Manager

In this section we will take a closer look at the [Result Manager](#) view. This will be done in the Executor perspective.

### The Result Manager

The MESSINA **Result Manager** is a view used for displaying the result of a **Test Case** or **Campaign** during and after execution. A tree structure/hierarchy is used to enable easy navigation through the different processes and the different levels of each process. Each execution process will create a new entry into the tree structure. The highest level of the structure identifies the **Test Case** or **Campaign**. The name used is generated automatically using the project name, testenvironment name, and a time stamp.

Expand the nodes in the **Result Manager** by clicking on the (+) next to each entry. Note that the name of the **Test Case** source code file (**MyTestCase.java** in our example) is listed for each entry in the **Result Manager** view. The parameters used for each **Test Case** execution are listed at the lowest level of the tree structure. A colour coding is used to allow us to quickly identify the test result (PASS/FAIL).

In the previous sections we executed our **Test Case** several times and we executed our **Campaign**. Remember that we created a Test Case where the condition of **air\_temp == air\_temp parameter** was not met (which was very easy because we only changed the **Parameter**). This resulted in the red entries in the **Result Manager** display.

Your **Result Manager** view might look something like this:



Name	Result	Date	Duration
*MyFirstProject_XiL_170224132007	PASSED (PASSED:1 FAILED:0 ERROR:0)		0:00:41.327
Project Parameters			
MyTestCase.java	PASSED	24.02.2017 13:20:07	0:00:41.327
*MyFirstProject_XiL_170224132244	FAILED (PASSED:0 FAILED:1 ERROR:0)		0:00:57.034
Project Parameters			
MyTestCase.java	Assertion failed: Temperature not reached (MyTestCase.java:33)	24.02.2017 13:22:44	0:00:57.034
*MyFirstProject_XiL_170224152135	FAILED (PASSED:1 FAILED:1 ERROR:0)		0:01:38.270
Project Parameters			
speed = 0			
timeout = 0			
air_temp = 0			
MyTestCampaign	FAILED (PASSED:1 FAILED:1 ERROR:0)		0:01:38.270
Campaign Parameters			
air_temp = 5			
timeout = 30000			
speed = 130			
MyTestCase.java (<default>)	PASSED	24.02.2017 15:21:35	0:00:41.366
Campaign Parameters			
air_temp = 5			
timeout = 30000			
speed = 130			
MyTestCase.java (<default>)	Assertion failed: Temperature not reached (MyTestCase.java:33)	24.02.2017 15:22:16	0:00:56.904
Campaign Parameters			
air_temp = 1			
timeout = 30000			
speed = 130			

The above picture shows three executions:

- The first is the execution of the **Test Case** where the parameters are set to allow a **Pass** result.
- The second execution of the **Test Case** set the **air\_temp** to a value that would not be reached. This causes the timeout condition and results in a **Fail** result.
- The third entry is the execution of the **Campaign** which executes the **Test Case** twice, but with 2 different sets of parameters (one has a **Pass** result, the other has a **Fail** result). Note that each execution of the **Test Case** within the **Campaign** results in a separate branch in the tree structure being created. This makes it very easy for us to follow the entire process step by step.

Each entry in the [Result Manager](#) will contain the **Name**, a test **Result**, the **Date**, and the **Duration**.

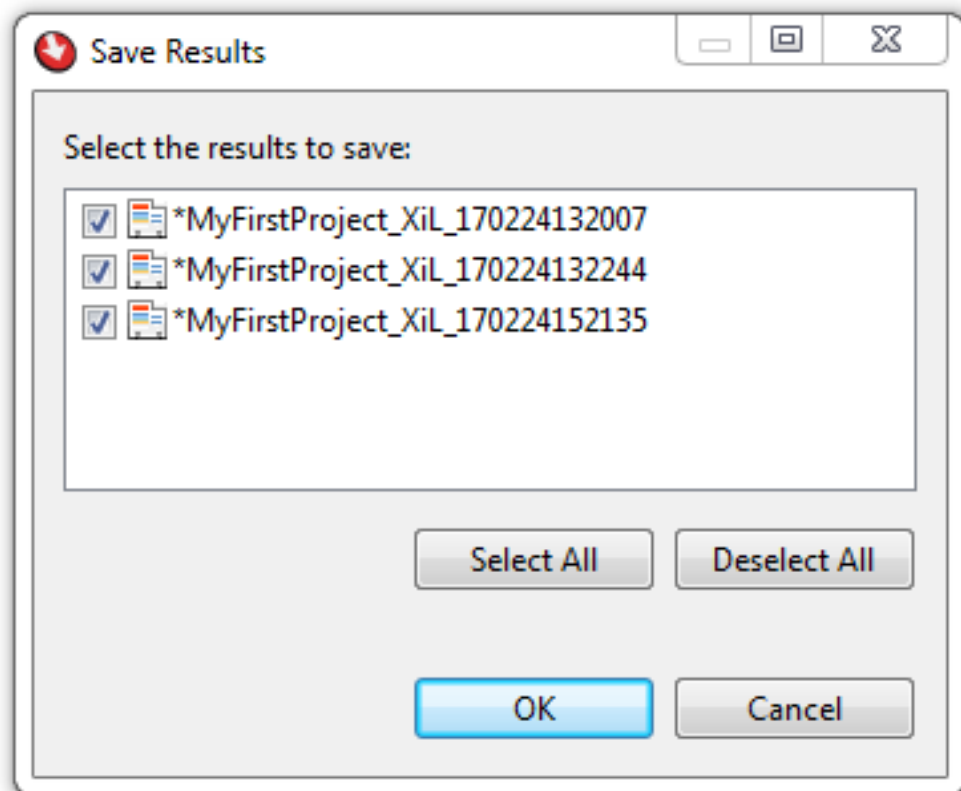
What can be done with the test results will be covered in the next steps.

## Step 13 - Test Results and Signal Traces

In this section we will take a closer look at how **Test Results** and **Signal Traces** are handled by MESSINA. The actions described here are all performed in the **Executor** perspective.

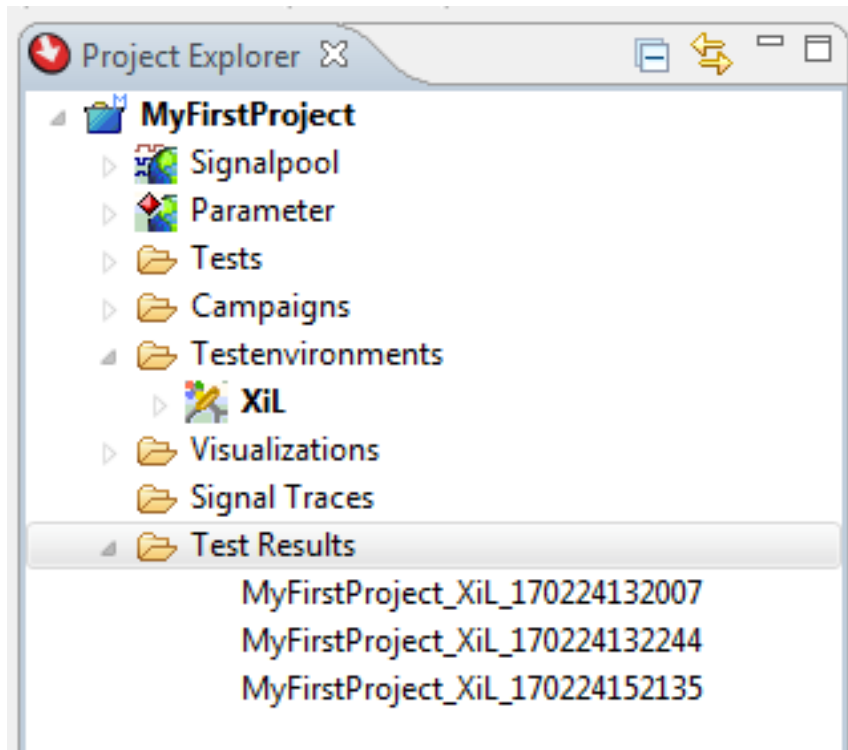
### Test Results

The **Test Results** folder shown in the [Project Explorer](#) gives a list of all results that were displayed in the [Result Manager](#) and saved. To save the test results you must first make sure that the **Result Manager** window has the focus. This can be done by clicking somewhere in the **Result Manager** window. Then press the **Save icon** in the menu bar (or **Main Menu** → **File** → **Save**). The following dialog box is called:



Select and press **OK** to save the results. In the next dialog you can add a comment to the test results, then press the **OK** button.

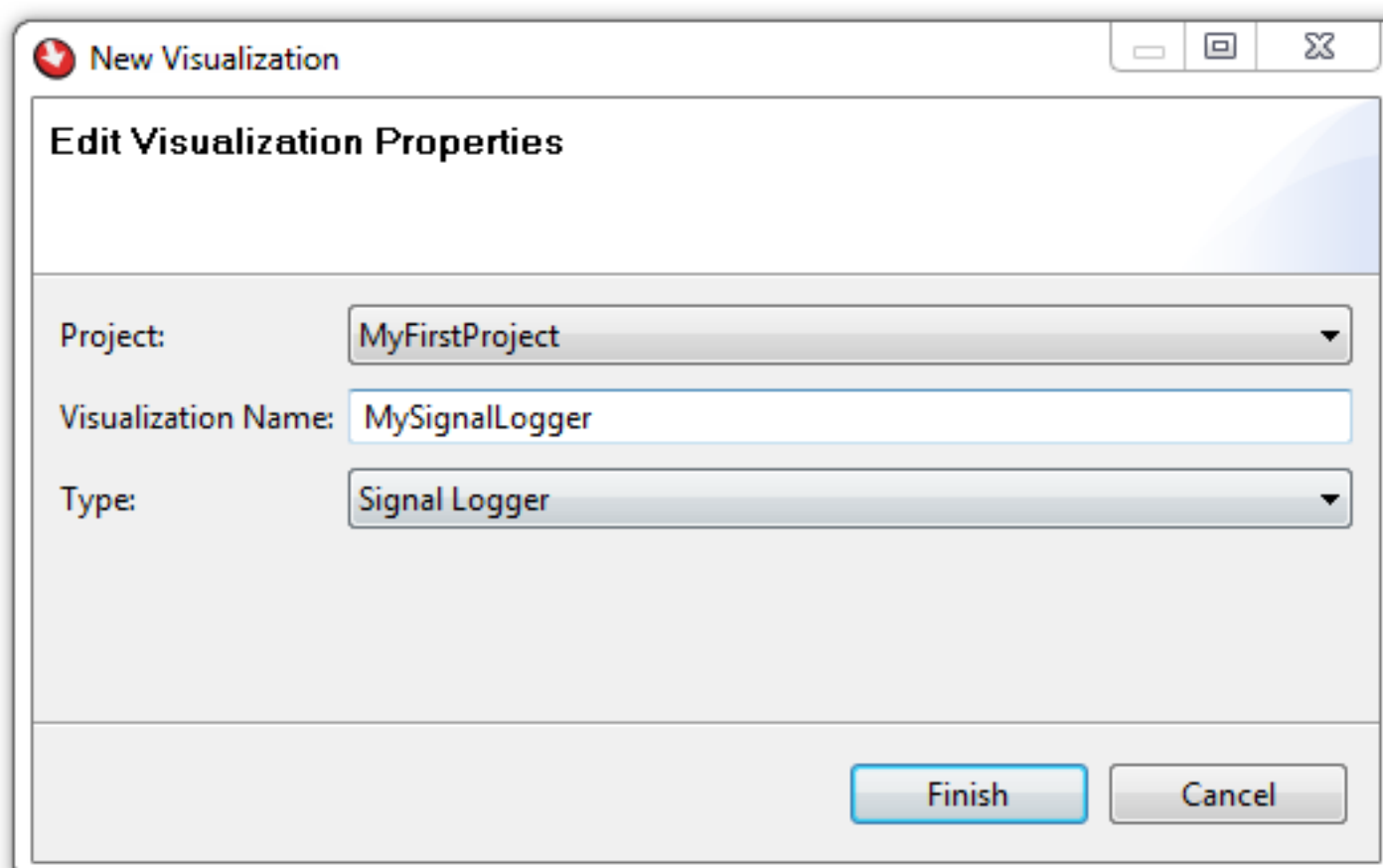
A new item is added to the **Test Results** folder in the [Project Explorer](#). This is the file where the test results are saved. The file name is generated automatically and includes a time stamp.



A new file can be generated for each set of test results as displayed in the [Result Manager](#). Each execution process (therefore each entry in the **Result Manager**) can be saved separately by marking it and selecting **Context Menu** → **Save**. Each execution process can also be repeated by selecting **Context Menu** → **Rerun**, which will result in a new execution process entry being made in the **Result Manager** view. An existing set of test results can be expanded by executing another process which will make another entry in the **Result Manager** window. The same test can be run repeatedly and saved after each execution if required.

### Signal Logging and Traces

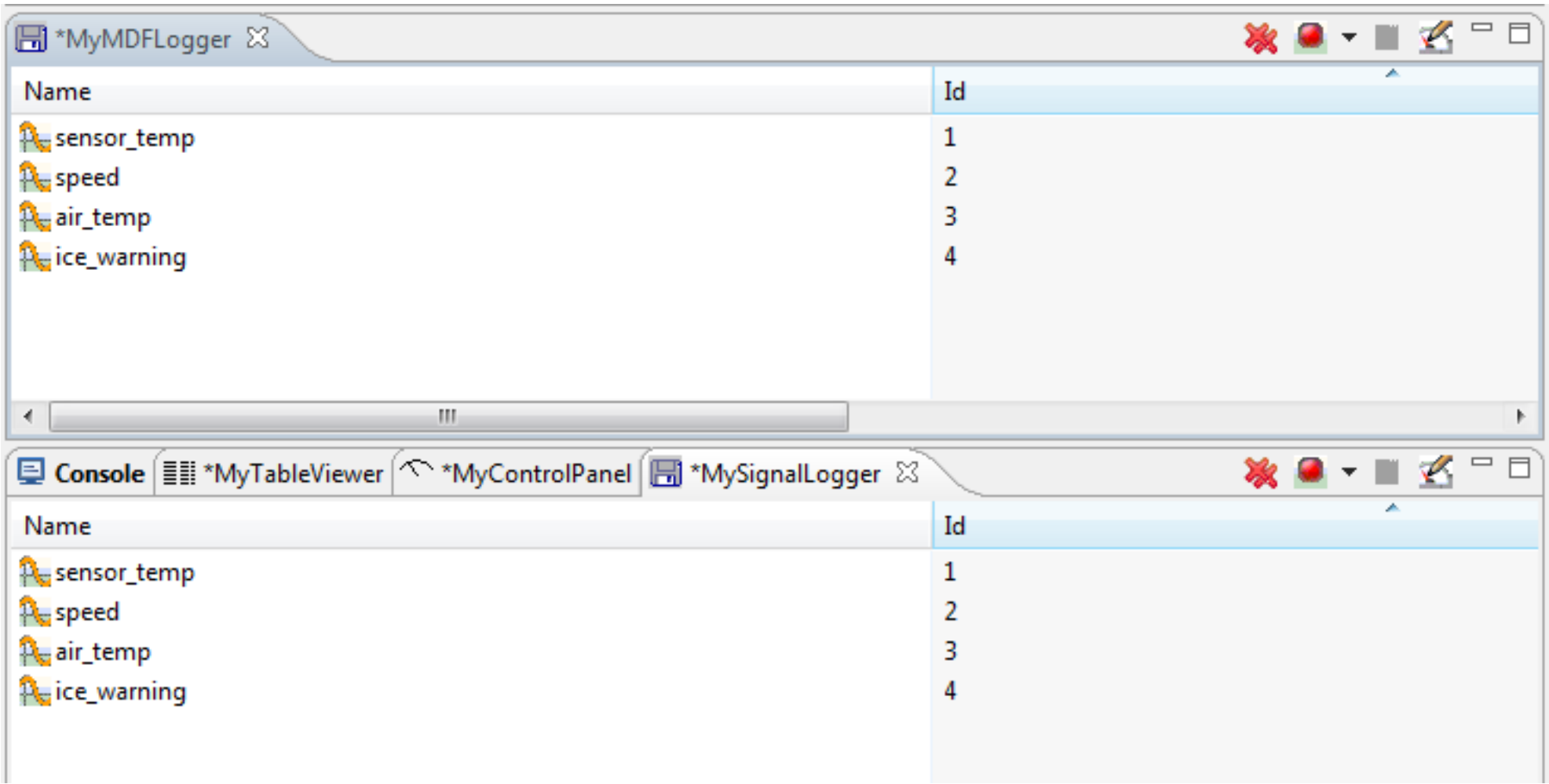
Signal [Logging](#) is performed by adding a **visualization** and selecting the **Signal Logger** or **MDF Logger** option from the visualization **Type** pull-down list.



Both of these log the selected signal data, but in a different format. The **Signal Logger** type stores the signal data in a standard ASCII format that is readable by any text editor program. MDF (Measurement Data Format) is commonly used in the automobile industry. The **MDF Logger** type stores the signal in the MDF format which can be used by external software for data viewing and analysis.

When a **Logging** process is being performed, MESSINA automatically adds an entry in the Signal Traces folder displayed in the **Project Explorer**. For each **Logging** process (in MDF and ASCII format), an entry is added. The entries shown here correspond to a data file which is created in the MESSINA project structure.


Signals are added to the logger visualizations the same way as for a table display or **Graph visualization**. Create 2 new visualizations, one a **Signal Logger** and the other an **MDF Logger**. Add all 4 signals from the **signalpool** to both the Signal Logger and the MDF Logger. The visualization view window should look like the picture below. The **SignalLog** and **MDF\_Log** visualizations both look the same, they simply list the signal that have been added to them.



Icons:

 Start Logging

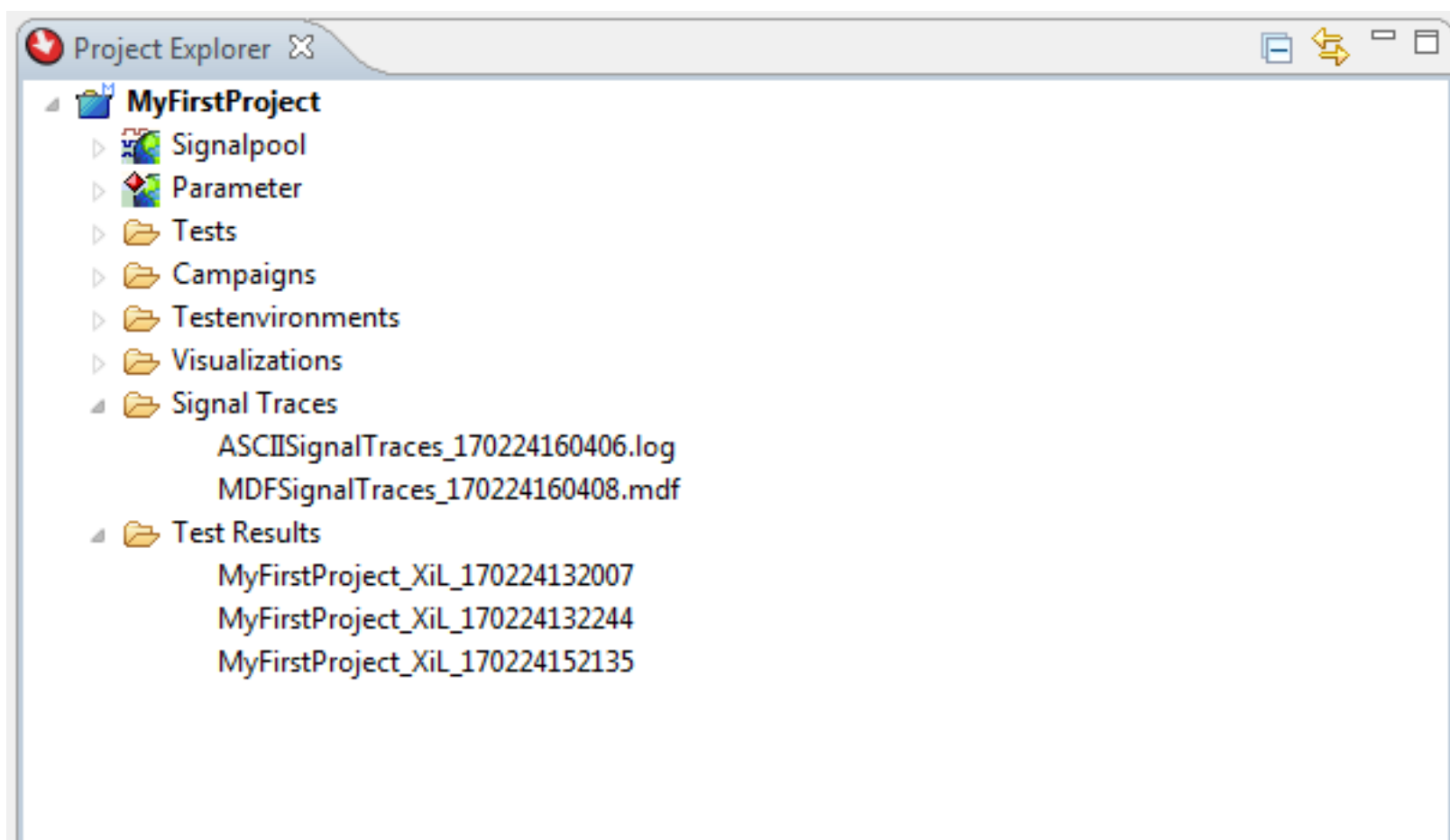
 Stop Logging

 Remove all signals

When the test is executed, a separate **Signal Traces** entry is made and a file will be created for the **SignalLog** and **MDF\_Log** containing the signal data and stored in the MESSINA project structure in a sub-folder called **Signal Traces**. For our example, the data file path would be:

- ...\MESSINA\tutorial\_workspace\MyFirstProject\Signal Traces\ASCIISignalTraces\_170224160406.log
- ...\MESSINA\tutorial\_workspace\MyFirstProject\Signal Traces\MDFSignalTraces\_170224160408.mdf

The file names are generated automatically and always include a time stamp. The [Project Explorer](#) now looks something like this:

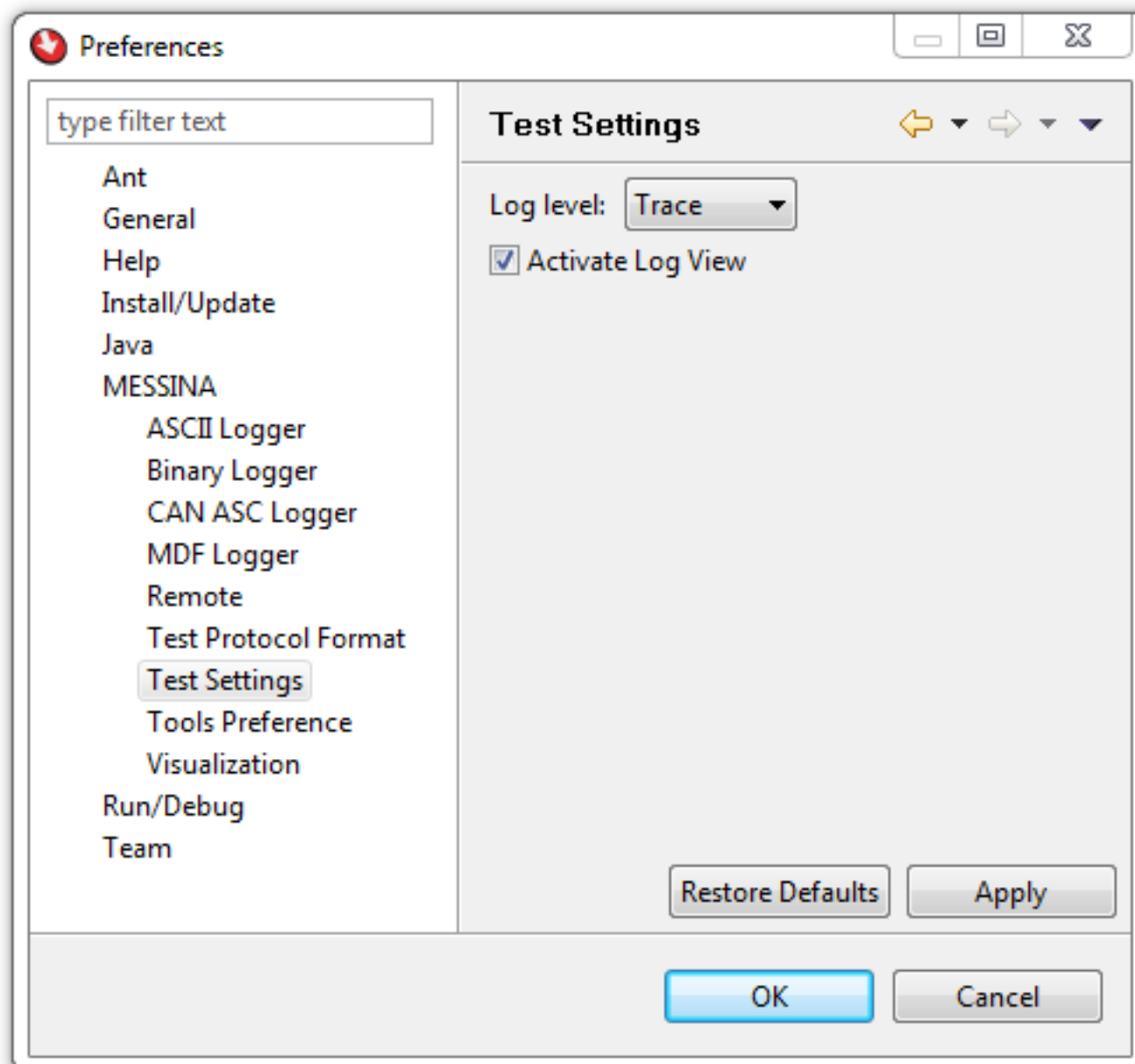


Double clicking an entry in the **Signal Traces** will call the standard viewer to display the data. The standard viewers must be set in the **Main Menu** → **Window** → **Preferences** section. For a detailed description of the Signal Trace files, refer to the [Traces and Logging](#) section of this documentation.

## Step 14 - The Log View and Log Levels

In this section we will take a closer look at the [Log View](#). This will be done in the Executor perspective.

It is possible to do a visual logging of the test process using the built-in test settings. This allows us to view logging information during a **Test Case** or **Campaign** execution process. Select **Main Menu** → **Window** → **Preferences** then select **Messina** → **Test Settings**. The following dialog is displayed:



The **Log level** pull-down list can be used to set a **log level**. Click on the pull-down list and select **Trace** to activate all logging operations. This setting will now be used as the default **log level** setting for all **Test Case** and **Campaign** executions.

Now, execute the **Campaign** for our **TempComp** example project using the **Context Menu** → **Execute 1x on default target** option. You will notice that the [Log View](#) automatically gets the focus in the visualization section of the **executor** view.

The **Log View** displays information based on the **Log Level** setting. There is one **Log View** for each target. In our case, this will be the SiL target. The logging information is written to the **Log View** continually during a running test process. The **TempComp** example **Campaign** was executed once



with the **Log Level** set to **Trace** to create the following output.

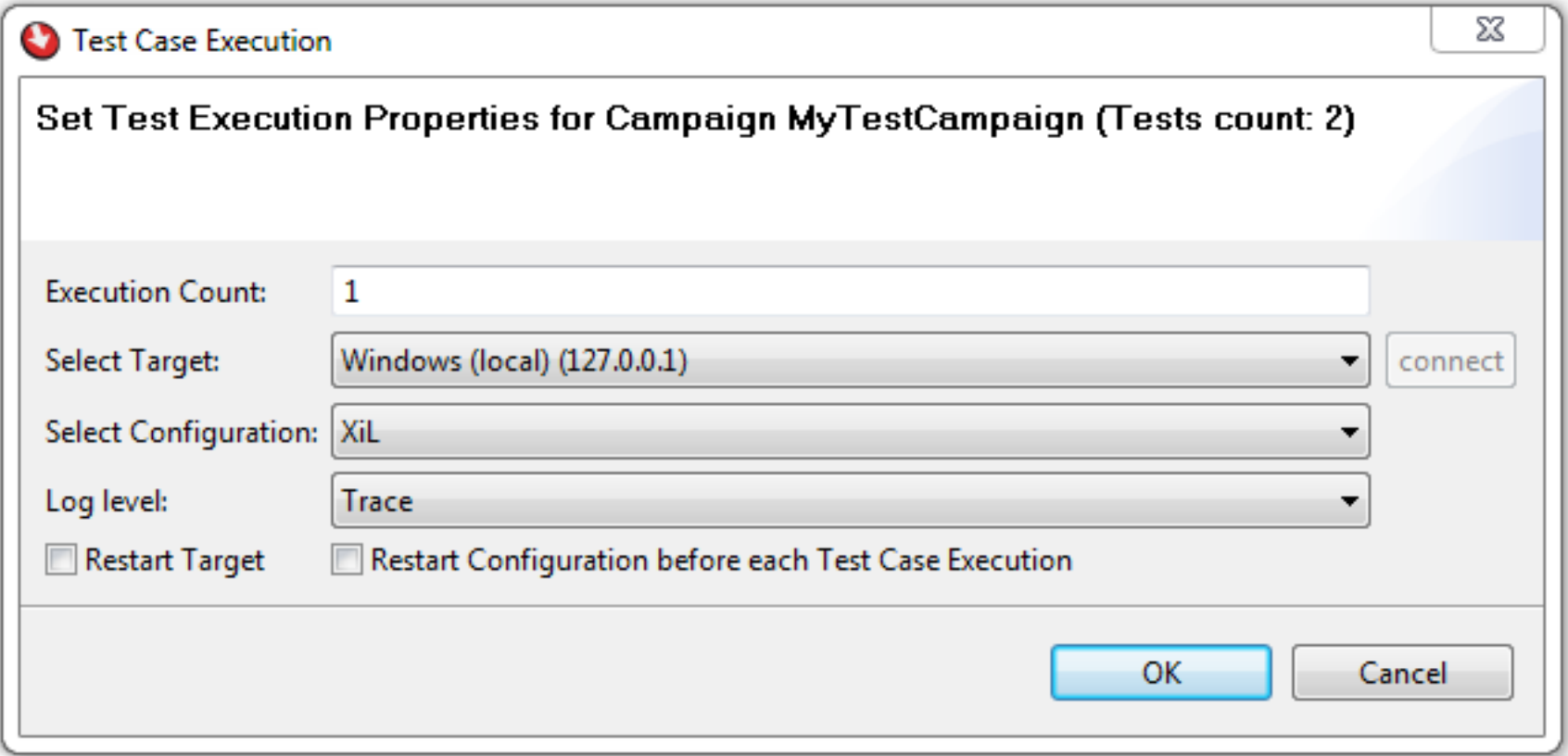
```

Console *MyControlPanel Windows (local)
0      Trace      Test Stage      Precondition
0      Trace      Set Value       speed (2): 0.0
0      Trace      Set Value       sensor_temp (1): 10.0
0      Trace      Wait Value      air_temp (3): 10.0, timeout: 60000
16615  Trace      Wait Ok         air_temp (3): 10.0
16615  Trace      Test Stage      Run
16615  Trace      Set Value       speed (2): 130.0
16615  Trace      Wait Value      air_temp (3): 5.0, timeout: 30000
28814  Trace      Wait Ok         air_temp (3): 5.0
28814  Trace      Test Stage      Postcondition
28814  Trace      Set Value       speed (2): 0.0
28814  Trace      Set Value       sensor_temp (1): 0.0
28814  Trace      Wait Value      air_temp (3): 0.0, timeout: 60000
41314  Trace      Wait Ok         air_temp (3): 0.0
41314  Info       Test Done       0
0      Trace      Test Stage      Precondition
0      Trace      Set Value       speed (2): 0.0
0      Trace      Set Value       sensor_temp (1): 10.0
0      Trace      Wait Value      air_temp (3): 10.0, timeout: 60000
16682  Trace      Wait Ok         air_temp (3): 10.0
16682  Trace      Test Stage      Run
16682  Trace      Set Value       speed (2): 130.0
16682  Trace      Wait Value      air_temp (3): 1.0, timeout: 30000
46682  Warning     Wait Failed     air_temp (3): 4.2
46682  Trace      Test Stage      Postcondition
46682  Trace      Set Value       speed (2): 0.0
46682  Trace      Set Value       sensor_temp (1): 0.0
46682  Trace      Wait Value      air_temp (3): 0.0, timeout: 60000
57084  Trace      Wait Ok         air_temp (3): 0.0
57084  Info       Test Done       -2000 (Assertion failed: Temperature not reached (MyTestCase.java:33))
  
```

The information displayed in the [Log View](#) is not saved to a file. It is intended as information to be displayed during a running test process. The **Log View** is a text display so we can mark, copy, and save the information to another file (e.g. to a text editor program) if required. The information displayed is currently limited to approx. 10k of data. Once this limit is reached, the oldest information is discarded when new information is added.

## Log Levels

Execute the **Campaign** again, but this time use the **Context Menu** → **Execute** option. The following dialog is called:



Notice that a **Log level** pull-down list is available and it is set to the option that we set in the **preferences** earlier. The **Log level** pull-down list can be used to override the default setting for this execution process only. Press **OK** and let the execution run again. Try changing the **Log level** and examine what effect this has on the output.

The **Log level** settings are cumulative. This means all options including and below the selected option are also active. For example, if the **Debug** option is selected, this would mean that the **Info**, **Warning**, **Error**, and **Fatal** options are also active. The following table summarizes the **log levels**:



Log Level	Test Case Constant Name	Description
<i>Trace</i>	LogLevel.TRACE	logs the stage (e.g. pre-condition, run, post-condition) and signal changes
<i>Debug</i>	LogLevel.DEBUG	User defined logging of messages and values in the Test Case (e.g. function call:  log(ITargetLogger.DEBUG, "hello")
<i>Info</i>	LogLevel.INFO	logs the test status (e.g. done) and the return value
<i>Warning</i>	LogLevel.WARNING	makes a log entry when a "wait" operation fails (e.g. waitTrigger, waitValue)
<i>Error</i>	LogLevel.ERROR	makes a log entry when an error occurs
<i>Fatal</i>	LogLevel.FATAL	makes a log entry when a fatal error (exceptions) occurs
<i>Off (default)</i>	-	nothing is logged

## Log Levels in Test Cases

As you can see from the table above, it is possible to access the **Log level** in the **Test Case** using either a discrete value or a pre-defined constant.

Switch to the **Designer** perspective and modify the **Test Case** by adding the following line of code in the run function just before the `return 0;` line:

```
air_temp.logValue(LogLevel.TRACE);
```

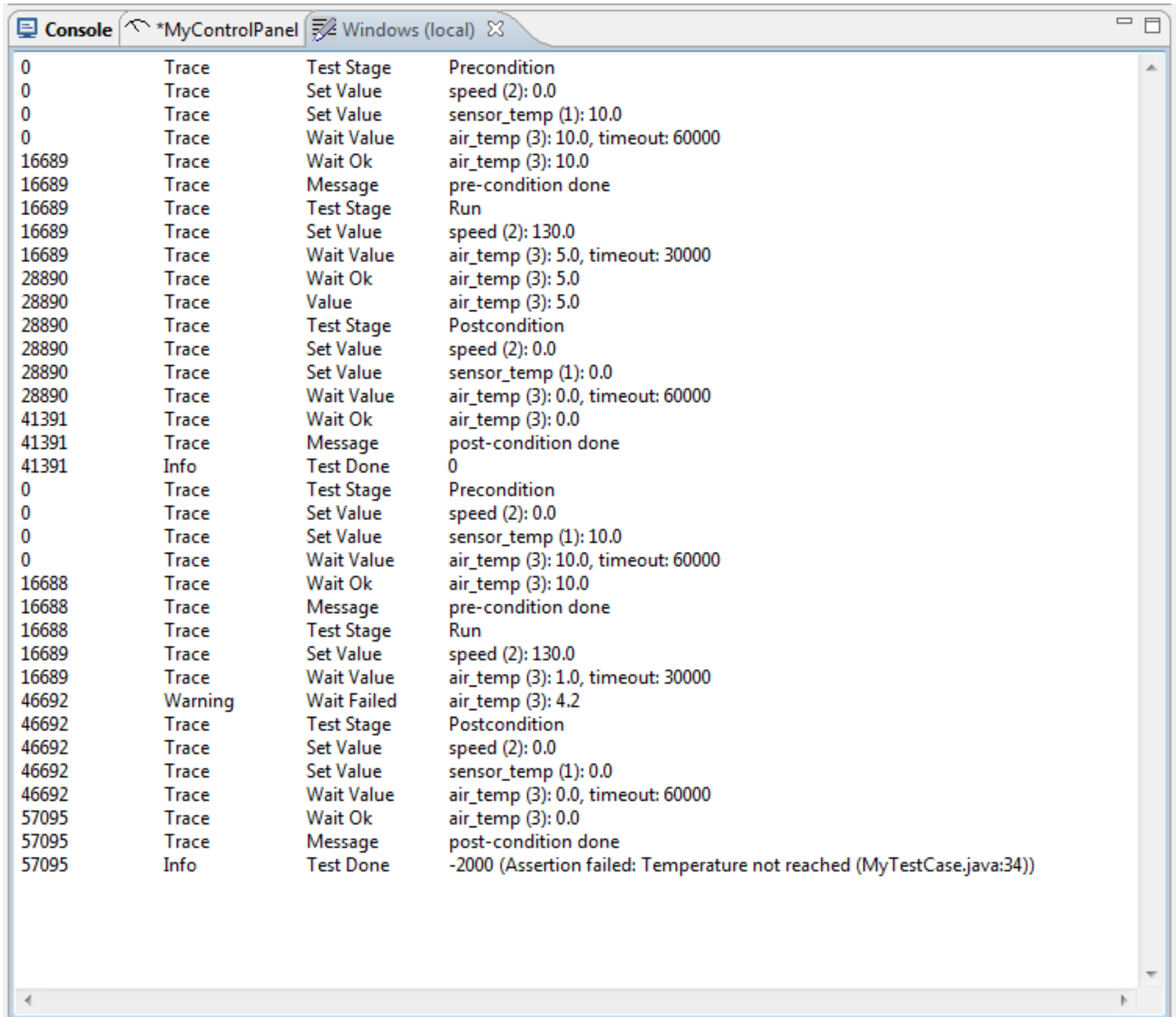
Now add the following line to the **preCondition** and **postCondition** functions respectively, again just before the return 0 line:

```
log(LogLevel.TRACE, "pre-condition done");
```

```
log(LogLevel.TRACE, "post-condition done");
```

These lines are designed to force an output to the **Log View** when the process is executed. The **Log Level** is set to **Trace** so that the value or message is always displayed.

Execute the **Campaign** again using the **Context Menu** → **Execute** option with the **Log Level** set to **Trace**. Examine the text in the [Log View](#) to confirm that each of the newly added log items is displayed. The **Log View** should look something like this:



# Step 15 - Test Reports

One requirement of any test system is always a [Test Report](#). Sometimes it is necessary to have information printable for reviewing or archiving if required. MESSINA handles this with a built-in report generator.

## Test Reports

In order to view a report, a style sheet must be assigned. This must be set in the **Main Menu** → **Window** → **Preferences** section. See the **Preferences** section of this documentation for a detailed description of how to set the **Preferences**.

**Test Reports** are created from **Test Results** that have been saved. To save the test results, select the Result Manager view by clicking in it and press the save icon. Double click the result file created in the previous steps (listed in the **Test Results** folder). A report that looks similar to the one shown below should appear in the Visualization view.

Test Report

File generated by user at 30.06.11 11:00:00

Project: MyFirstProject

Configuration: XiL

	TempComp_Windows	(Matlab Simulink)	3.1.0.10986 (1.229)
--	------------------	-------------------	---------------------

Target:

	Windows (local)	(Windows)	3.1.0.10986 / Windows 5.1 (3.0) 1	VM	3.1.0.10986
--	-----------------	-----------	-----------------------------------	----	-------------

Comments:

Statistics

Successful		50,00 %
------------	--	---------

Failures		50,00 %
Errors		0,00 %
Inconclusive		0,00 %
Tests	2	

Successful Tests

Test name	Test group	Time of test	Duration	Comments
MyTestCase.java (<default>)	/MyTestCampaign	30.06.2011 21:11:55	0:00:41.579	

Go to top

Failed tests

Test name	Test group	Time of test	Duration	Comments
MyTestCase.java (<default>)	/MyTestCampaign	30.06.2011 21:12:37	0:00:57.095	Assertion failed: Temperature not reached (MyTestCase.java:14)

Go to top

Error tests

No error tests.

Go to top

Test Results Info  
1 MyTestCampaign

Result:	FAILED
Start:	30.06.2011 21:11:55
Duration:	0:01:38.674
Comments:	Total 2; Successful 1

The format of the report is standardized so it will always have the same layout.