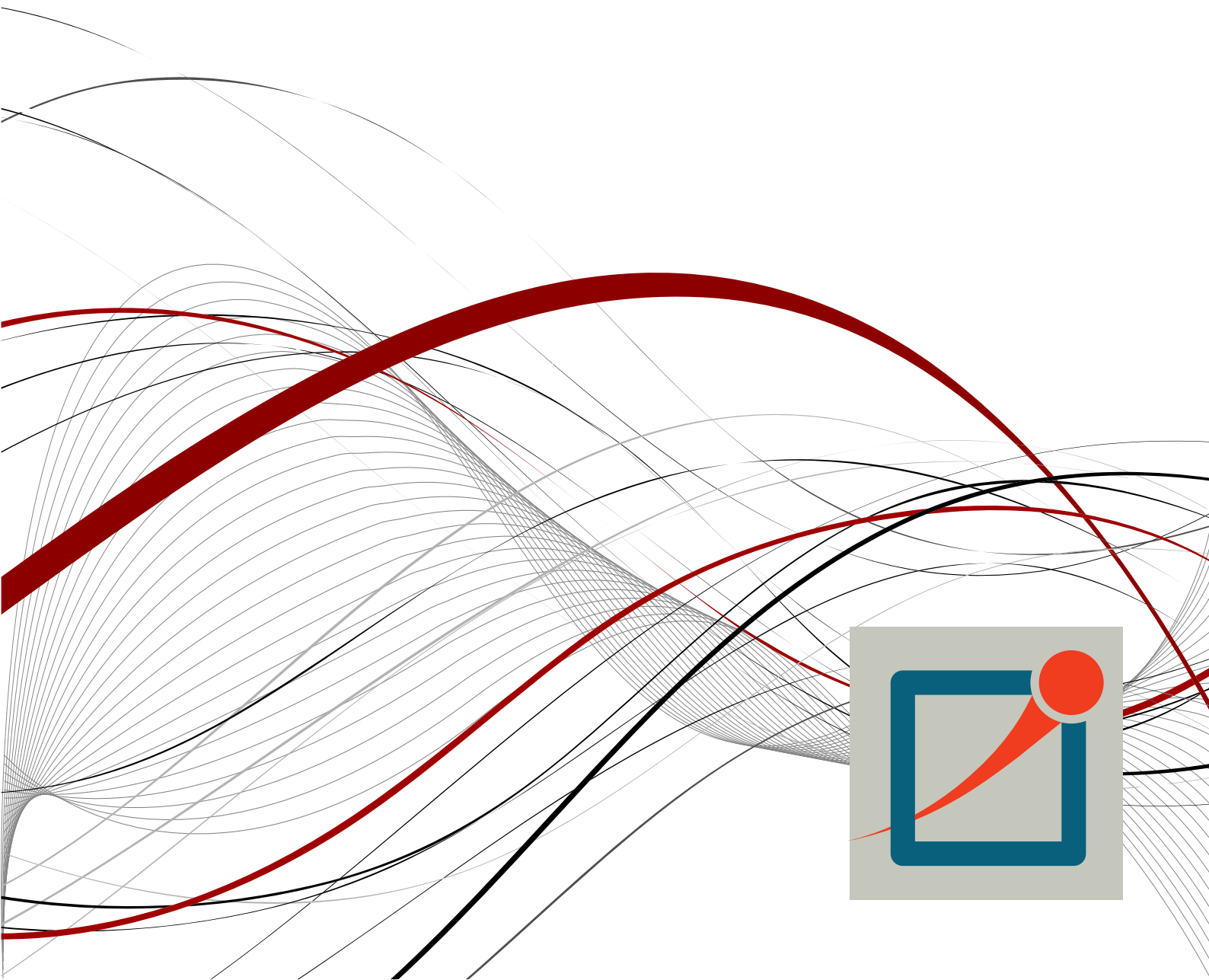
OMNEST

User Guide

Version 6.1



Build: 241025-2758eec6d6

CONTENTS

1 Introduction **1**
- 1.1 The Workbench . . . 1
- 1.2 Workspaces . . . 2
- 1.3 The Simulation Perspective . . . 3
- 1.4 Configuring OMNEST Preferences . . . 3
- 1.5 Creating OMNEST Projects . . . 4
- 1.6 Project References . . . 4
- 1.7 Getting Help . . . 5

2 Editing NED Files **7**
- 2.1 Overview . . . 7
- 2.2 Opening Older NED Files . . . 7
- 2.3 Creating New NED Files . . . 8
- 2.4 Using the NED Editor . . . 9
- 2.5 Associated Views . . . 17

3 Editing INI Files **21**
- 3.1 Overview . . . 21
- 3.2 Creating INI Files . . . 21
- 3.3 Using the INI File Editor . . . 22
- 3.4 Associated Views . . . 27

4 Editing Message Files **31**
- 4.1 Creating Message Files . . . 31
- 4.2 The Message File Editor . . . 32

5 C++ Development **33**
- 5.1 Introduction . . . 33
- 5.2 Prerequisites . . . 33
- 5.3 Creating a C++ Project . . . 34
- 5.4 Editing C++ Code . . . 36
- 5.5 Building the Project . . . 40
- 5.6 Configuring the Project . . . 42
- 5.7 Project Features . . . 49
- 5.8 Project Files . . . 53

6 Launching and Debugging **55**
- 6.1 Introduction . . . 55
- 6.2 Launch Configurations . . . 56
- 6.3 Running a Simulation . . . 56
- 6.4 Batch Execution . . . 60
- 6.5 Debugging a Simulation . . . 61
- 6.6 Just-in-Time Debugging . . . 63
- 6.7 Profiling a Simulation on Linux . . . 63
- 6.8 Controlling the Execution and Progress Reporting . . . 63

7 The Qtenv Graphical Runtime Environment **67**
7.1 Features . . . 67
7.2 Overview of the User Interface . . . 68
7.3 Using Qtenv . . . 69
7.4 Using Qtenv with a Debugger . . . 75
7.5 Parts of the Qtenv UI . . . 75
7.6 Inspecting Objects . . . 83
7.7 The Preferences Dialog . . . 86
7.8 Qtenv and C++ . . . 93
7.9 Reference . . . 94

8 Sequence Charts **97**
8.1 Introduction . . . 97
8.2 Creating an Eventlog File . . . 97
8.3 Sequence Chart . . . 98
8.4 Eventlog Table . . . 105
8.5 Filter Dialog . . . 108
8.6 Other Features . . . 110
8.7 Examples . . . 112

9 Analyzing the Results **121**
9.1 Overview . . . 121
9.2 Creating Analysis Files . . . 122
9.3 Opening Older Analysis Files . . . 123
9.4 Using the Analysis Editor . . . 123
9.5 The Inputs Page . . . 123
9.6 The Browse Data Page . . . 125
9.7 The Charts Page . . . 128
9.8 The Outline View . . . 128
9.9 Basic Chart Usage . . . 129
9.10 Configuring Charts . . . 134
9.11 Editing the Chart Script . . . 143
9.12 Editing Dialog Pages . . . 145
9.13 Chart Programming . . . 147
9.14 Custom Chart Templates . . . 150
9.15 Under the Hood . . . 152

10 NED Documentation Generator **153**
10.1 Overview . . . 153

11 Extending the IDE **157**
11.1 Installing New Features . . . 157
11.2 Adding New Wizards . . . 157
11.3 Project-Specific Extensions . . . 158

CHAPTER

ONE

INTRODUCTION

The OMNEST simulation IDE is based on the Eclipse platform and extends it with new editors, views, wizards, and other functionality. OMNEST adds functionality for creating and configuring models (NED and INI files), performing batch executions, and analyzing the simulation results, while Eclipse provides C++ editing, SVN/GIT integration, and other optional features (UML modeling, bug-tracker integration, database access, etc.) via various open-source and commercial plug-ins. The environment will be instantly recognizable to those familiar with the Eclipse platform.

1.1 The Workbench

The Eclipse main window consists of various Views and Editors. These are collected into Perspectives that define which Views and Editors are visible and how they are sized and positioned.

Eclipse is a very flexible system. You can move, resize, hide, and show various panels, editors, and navigators. This allows you to customize the IDE to your liking, but it also makes it more difficult to describe. First, we need to make sure that we are looking at the same thing.

The OMNEST IDE provides a "Simulation perspective" to work with simulation-related NED, INI, and MSG files. To switch to the simulation perspective, select *Window* → *Open Perspective* → *Simulation*.

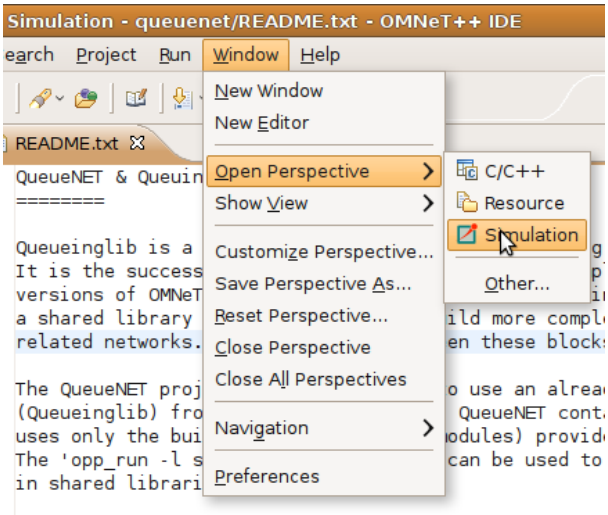


Fig. 1.1: Selecting the "Simulation Perspective" in Eclipse

Most interface elements within Eclipse can be moved or docked freely, so you can construct your own workbench to fit your needs.

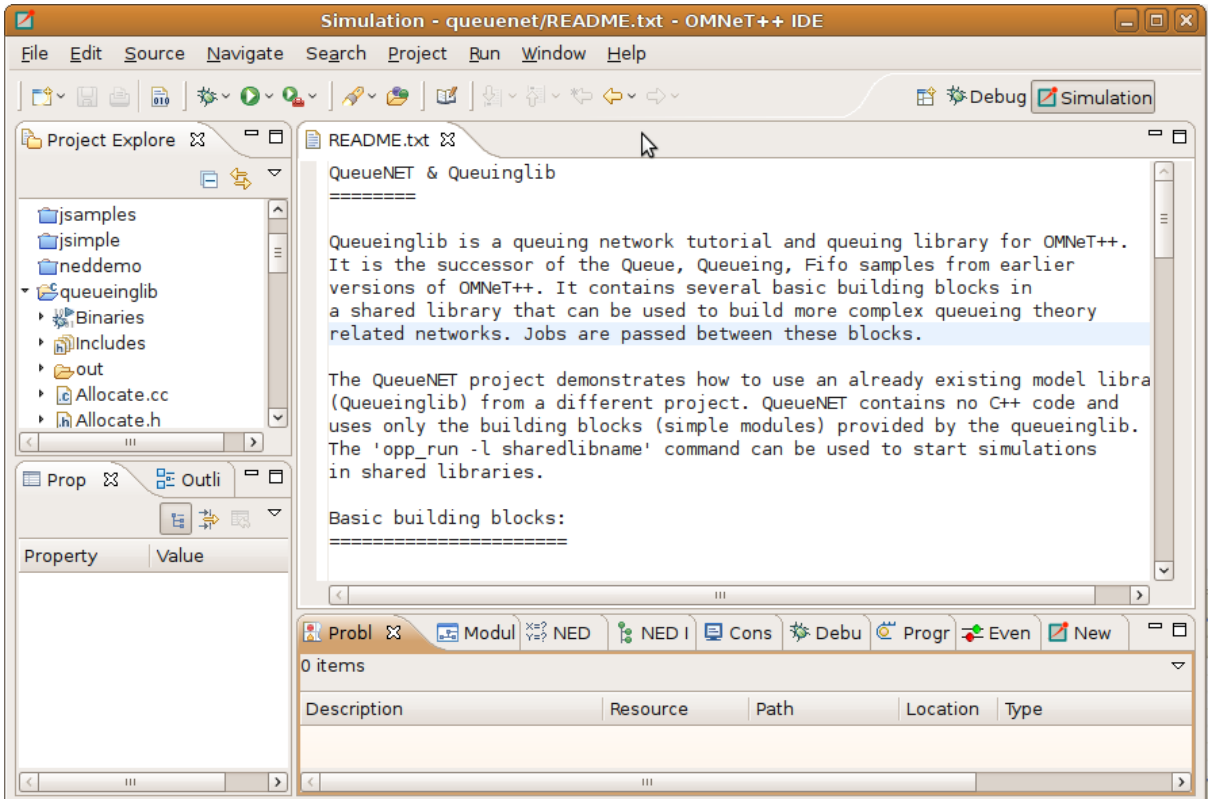


Fig. 1.2: Default layout of the OMNEST IDE

The *Project Explorer* on the top left part of the screen shows the projects and their content in your workspace. In the example above, the "queueinglib" demo project is open. You can see the various `.ned`, `.ini`, and other files inside. A number of views are docked at the bottom of the window.

The screenshot shows the open README.txt file in the editor area. When a user double-clicks on a file, Eclipse automatically launches the editor associated with that particular file type.

The *Properties View* contains information on the particular object that is selected in the editor area or one of the other views that serve as a selection provider. The *Problems View* references code lines where Eclipse encountered problems.

Several OMNEST-specific views exist that can be used during development. We will discuss how you can use them effectively in a later chapter. You can open any View by selecting *Window* → *Show View* from the menu.

1.2 Workspaces

A workspace is basically a directory where all your projects are located. You may create and use several workspaces and switch between them as needed. During the first run, the OMNEST IDE offers to open the samples directory as the workspace, so you will be able to experiment with the available examples immediately. Once you start working on your own projects, we recommend that you create your own workspace by selecting *File* → *Switch Workspace* → *Other*. You can switch between workspaces as necessary. Please be aware that the OMNEST IDE restarts with each switch in workspaces. This is normal. You can browse workspace content in the *Project Explorer*, *Navigator*, *C/C++ Projects*, and similar views. We recommend using the *Project Explorer*.

1.3 The Simulation Perspective

The OMNEST IDE defines the *Simulation Perspective* so that it is specifically geared towards the design of simulations. The *Simulation Perspective* is simply a set of conveniently selected views, arranged to make the creation of NED, INI, and MSG files easier. If you are working with INI and NED files a lot, we recommend selecting this perspective. Other perspectives are optimized for different tasks like C++ development or debugging.

1.4 Configuring OMNEST Preferences

The OMNEST IDE preferences dialog is available through the standard preferences menu, which is under the main Window menu item. These settings are global and shared between all projects. The OMNEST install locations are automatically filled in for you after installation. The default settings for the NED documentation generation assume that the PATH environment variable is already set so that third-party tools can be found. The license configuration settings specify the preferred license type or custom license text. The IDE will copy the license into new files and projects. The license will also be shown in the generated NED documentation.

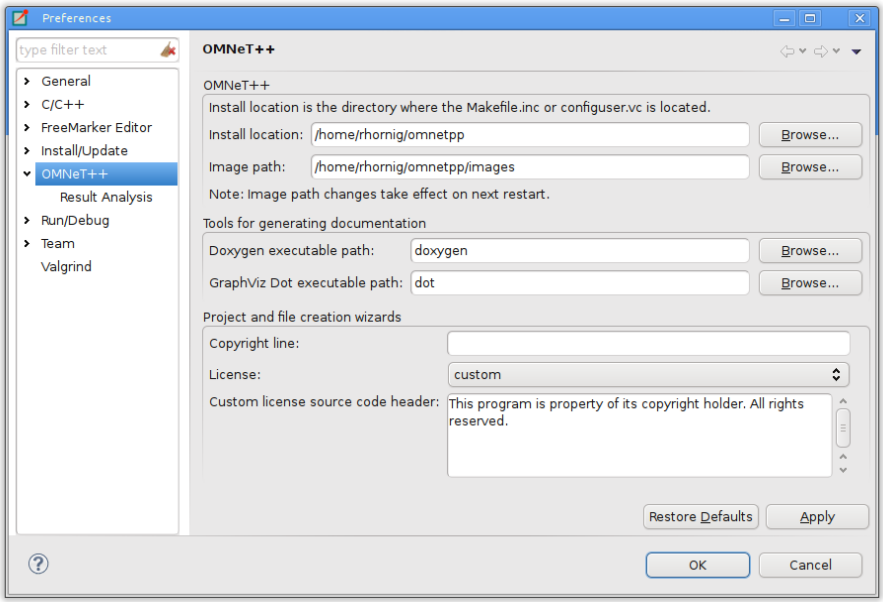


Fig. 1.3: Configuring OMNEST preferences

Use the Browse buttons to find files or folders easily. Specify the full path for executables if you do not want to extend the PATH environment variable.

1.5 Creating OMNEST Projects

In Eclipse, all files are within projects, so you will need a suitable project first. The project needs to be one designated as an OMNEST Project (in Eclipse lingo, it should have the OMNEST Nature). The easiest way to create such a project is to use a wizard. Choose *File* → *New* → *OMNEST Project* from the menu, specify a project name, and click the *Finish* button. If you do not plan to write simple modules, you may unselect the *C++ Support* checkbox, which will disable all C++ related features for the project.

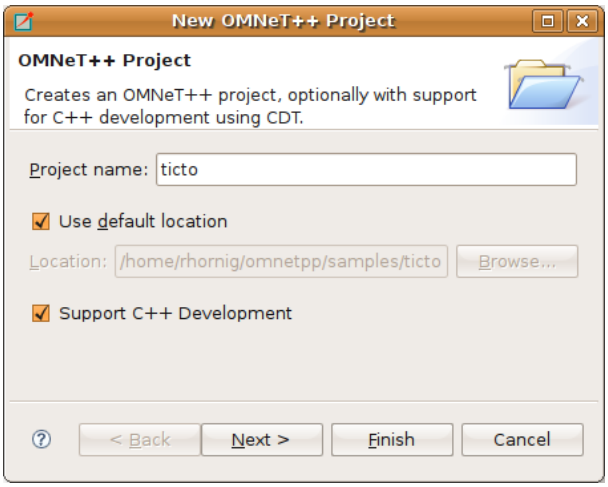


Fig. 1.4: Creating a new OMNEST project

1.6 Project References

Most aspects of a project can be configured in the *Project Properties* dialog. The dialog is accessible via the *Project* → *Properties* menu item or by right-clicking the project in the *Project Explorer* and choosing *Properties* from the context menu.

An important Eclipse concept is that a project may reference other projects in the workspace; project references can be configured in the *Project References* page of the properties dialog. To update the list of referenced projects, simply check those projects in the list that your project depends on, then click *Apply*. Note that circular references are not allowed (i.e. the dependency graph must be a tree).

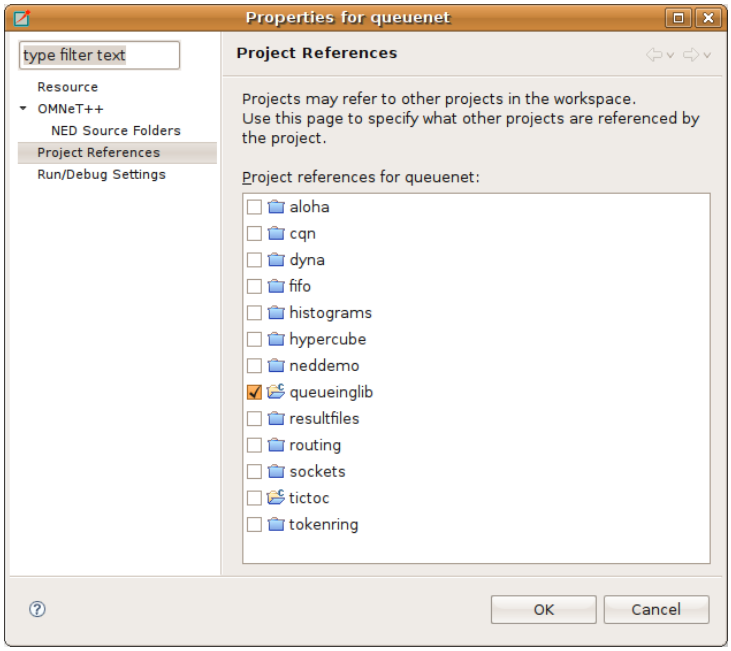


Fig. 1.5: Setting project dependencies

In the OMNEST IDE, all NED types, C++ code, and build artifacts (executables, libraries) in a project are available to other projects that reference the given project.

Note: To see an example of project references, check the "queuenet" and "queueinglib" example projects. In this example, "queuenet" references "queueinglib". "Queueinglib" provides simple modules (NED files, and a prebuilt shared library that contains the code of the simple modules) and makes those modules available to "queuenet" that contains simulations (networks and ini files) built from them.

1.7 Getting Help

You may access the online help system from the *Help* → *Help Contents* menu item. The OMNEST IDE is built on top of Eclipse, so if you are not familiar with Eclipse, we recommend reading the *Workbench User Guide* and the *C/C++ Development User Guide* before starting to use OMNEST-specific features.

CHAPTER

TWO

EDITING NED FILES

2.1 Overview

When you double-click a `.ned` file in the IDE, it opens in the NED editor. The new NED editor is a dual-mode editor. In the graphical mode, you can edit the network using the mouse. The textual mode allows you to work directly on the NED source.

When the IDE detects errors in a NED file, the problem will be flagged with an error marker in the *Project Explorer*, and the *Problems View* will show the description and location of the problem. Additionally, error markers will appear in the text window or on the graphical representation of the problematic component. Opening a NED file with an error will open it in text mode. Switching to graphical mode is only possible if the NED file is syntactically correct.

Note: As a side effect, if there are two modules with the same name and package in related projects, they will collide, and both will be marked with an error. Furthermore, the name will be treated as undefined, and any other modules depending on it will also generate an error (thus, a "no such module type" error may mean that there are actually multiple definitions which nullify each other).

2.2 Opening Older NED Files

The syntax of NED files has significantly changed from the 3.x version. The NED editor primarily supports the new syntax. However, it is still possible to read and display NED files with the old syntax. It is important to note that many of the advanced features (syntax highlighting, content assistance, etc.) will not work with the old syntax. There is automatic conversion from the old syntax to the new, available both from the NED editor and as an external utility program (**`opp_nedtool`**).

The `gned` program from OMNEST 3.x viewed NED files in isolation. In contrast, the OMNEST IDE gathers information from all `.ned` files in all open OMNEST projects and makes this information available to the NED editor. This is necessary because OMNEST 4.x modules may inherit parameters, visual appearance, or even submodules and connections from other modules. So, it is only possible to display a compound module correctly if all related NED definitions are available.

2.3 Creating New NED Files

Once you have an empty OMNEST project, you can create new NED files. Choose *File* → *New* → *Network Description File* from the menu. A wizard will appear where you can specify the target directory and the file/module name. You may choose to create an empty NED file, a simple/compound module, or a network. When you press the *Finish* button, a new NED file will be created with the requested content.

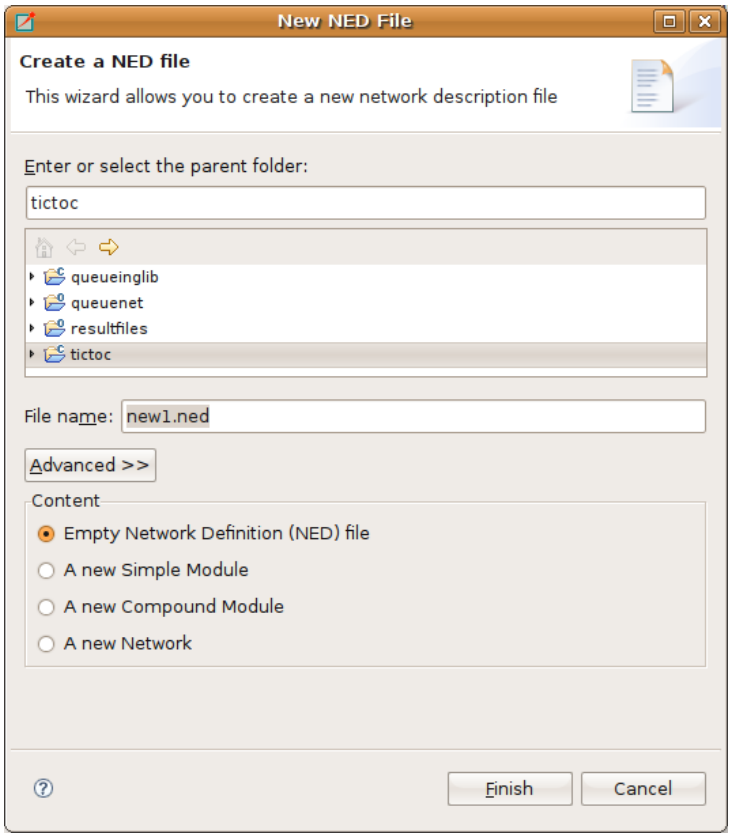


Fig. 2.1: Creating a new NED file

Tip: Make sure that the NED file and the contained module have the same name. For example, a compound module named `Wireless42` should be defined within its own `Wireless42.ned` file.

2.3.1 NED Source Folders

You can specify the folders the IDE should scan for NED files and use as the base directory for your NED package hierarchy. The IDE will not use any NED files outside the specified NED Source Folders, and those files will be opened in a standard text editor. To specify the directory where the NED files will be stored, right-click on the project in the *Project Explorer* and choose *Properties*. Select the *OMNEST* → *NED Source Folders* page and click on the folders where you store your NED files. The default value is the project root.

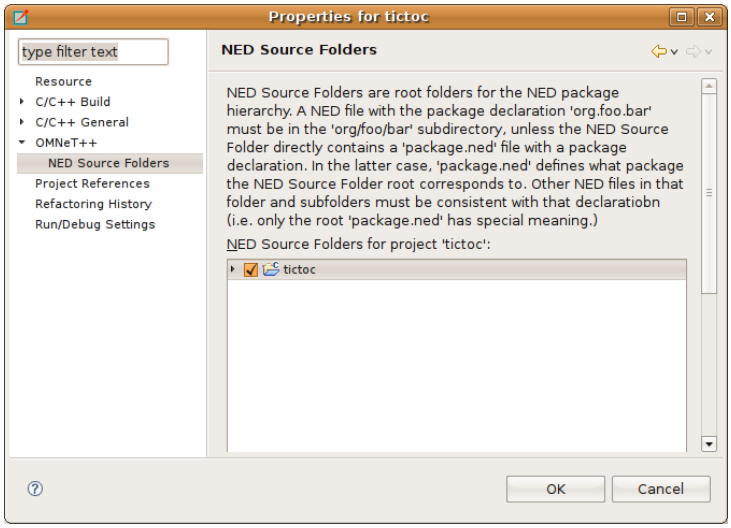


Fig. 2.2: Specifying the folder where NED files will be stored

2.4 Using the NED Editor

To open an NED file, double-click its icon in the *Project Explorer*. If the NED file can be parsed without an error, the graphical representation of the file opens; otherwise, the text view opens, and the text is annotated with error markers.

Warning: Only files located in NED Source Folders open with the graphical editor. If a NED file is not in the NED Source Folders, it opens in a standard text editor.

You can switch between graphical and source editing modes by clicking the tabs at the bottom of the editor or by using the `Alt+PGUP` / `Alt+PGDN` key combinations. The editor tries to keep the selection during the switch. Selecting an element in a graphical view and then switching to text view will move the cursor to the related element in the NED file. When switching back to the graphical view, the graphical editor tries to select the element that corresponds to the cursor's location in the NED source. This allows you to keep the context, even when switching back and forth.

2.4.1 Editing in Graphical Mode

The graphical editor displays the visible elements of the loaded NED file. Simple modules, compound modules, and networks are represented by figures or icons. Each NED file can contain more than one module or network. If it does, the corresponding figures appear in the same order as they are found in the NED file.

Tip: Place only a single module or network into an NED file and name the file according to the module name.

Simple modules and submodules are represented as icons while compound modules and networks are displayed as rectangles where other submodules can be dropped. Connections between submodules are represented either by lines or arrows depending on whether the connection was uni- or bi-directional. Submodules can be dragged or resized using the mouse and connected using the Connection Tool in the palette.

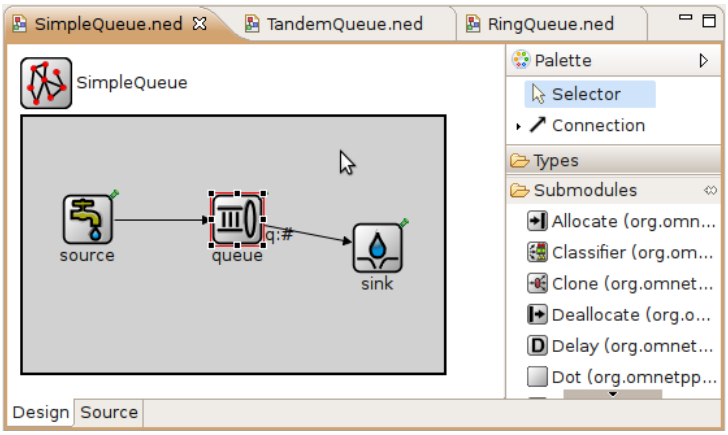


Fig. 2.3: Graphical NED Editor

The palette is normally on the right side of the editor area. The upper part of the palette contains the basic tools: selector, connection selector, and the connection creator tool. To use a palette item, click on it, and click in the module where you want to place/activate it. The mouse pointer gives you feedback whether the requested operation is allowed. The middle part of the toolbox contains the basic elements that can be placed at the top level in an NED file (simple module, compound module, interface, channel, etc.) and a "generic" submodule. Click on any of these and then click into the editor area to create an instance. The bottom part of the palette contains all module types that can be instantiated as a submodule. They are shortcuts for creating a generic submodule and then modifying its type. They display the default icon (if any) and a short description if you hover the mouse over them. You can configure the palette by right-clicking a button and selecting *Settings* or filter its content by selecting *Select Packages*

Right-clicking any element in the edited NED file brings up a context menu that allows various actions like changing the icon, pinning/unpinning a submodule, re-layouting a compound module, or deleting/renaming the element. There are also items to activate various views. For example, the *Properties View* allows you to edit properties of the element.

Hovering over an element displays its documentation (the comment in the NED source above the definition) as a tooltip. Pressing `F2` makes the tooltip window persistent, so it can be resized and scrolled for more convenient reading.

Creating Modules

To create a module or a submodule, click on the appropriate palette item and then click where you want to place the new element. Submodules can only be placed inside compound modules or networks.

Creating Types and Inner Types

To create a type or an inner type inside a compound module, click on the appropriate palette item in the "Types" drawer and then click where you want to place the new element. If you click on the background, a new top-level type will be created. Clicking on an existing compound module or network creates an inner type inside that module.

Creating and Changing Connections

Select the *connection tool* (if there are channels defined in the project, you can use the drop-down to select the connection channel type). First, click the source module, and then the destination. A popup menu will appear, asking which gates should be connected on the two selected modules. The tool only offers valid connections (e.g., it will not offer to connect two output gates).

Reconnecting Modules

Clicking and dragging a connection endpoint to another module will reconnect it (optionally, asking which gate should be connected). If you only want to change the gate, drag the connection endpoint, and drop it over the original module. A popup will appear asking for the source or destination gate.

Selecting Elements

You can select an element by clicking on it or by dragging a rectangle over the target modules. A compound module can be selected by clicking on its border or title. If you only want to select connections within a selection rectangle, use the *connection selector* tool in the dropdown menu of the *connection tool* . The `Ctrl` and `Shift` keys can be used to add/remove the current selection. Note that the keyboard (arrow keys) can also be used to navigate between submodules. You can also select using a selection rectangle by dragging the mouse around the modules.

Undo, Redo, Deleting Elements

Use `Ctrl+Z` and `Ctrl+Y` for undo and redo, respectively, and the `DEL` key for deletion. These functions are also available in the *Edit* menu and in the context menu of the selected element.

Moving and Resizing Elements

You can move/resize the selected elements with the mouse. Holding down `Shift` during move will perform a constrained (horizontal, diagonal, or vertical) move operation. `Shift` + resize will keep the aspect ratio of the element.

If you turn on *Snap to Geometry* in the *View* menu, helper lines will appear to align with other modules. Selecting more than one submodule activates the *Alignment* menu (found in both the *View* menu and the context menu).

Copying Elements

Holding down `Ctrl` while dragging will clone the module(s). Copy/Paste can also be used on single modules and with group selection.

Zooming

Zooming in and out is possible from the *View* menu or using `Ctrl+-`, `Ctrl+=`, or holding down `Ctrl` and using the mouse wheel.

Pinning, Unpinning, Re-Layouting

A submodule display string may or may not contain explicit coordinates for the submodule; if it does not, then the location of the submodule will be determined by the layouting algorithm. A submodule with explicit coordinates is pinned; one without is unpinned. The Pin action inserts the current coordinates into the display string, and the Unpin action removes them. Moving a submodule also automatically pins it. The position of an unpinned module is undetermined and may change every time the layouting algorithm runs. For convenience, the layouter does not run when a submodule gets unpinned (so that the submodule does not jump away on unpinning), but this also means that unpinned submodules may appear at different locations the next time the same NED file is opened.

Changing a Module Property

To change a module property, right-click on it and select the *Properties* menu item from the context menu or select the module and modify that property in the *Properties View*. Alternatively, you can press `Ctrl+Enter` when the module is selected. NED properties like name, type, and vector size are available on the *General* tab. Visual properties like icon, size, color, border, etc. can be set on the *Appearance* tab. You can check how your module will look in the preview panel at the bottom of the dialog.

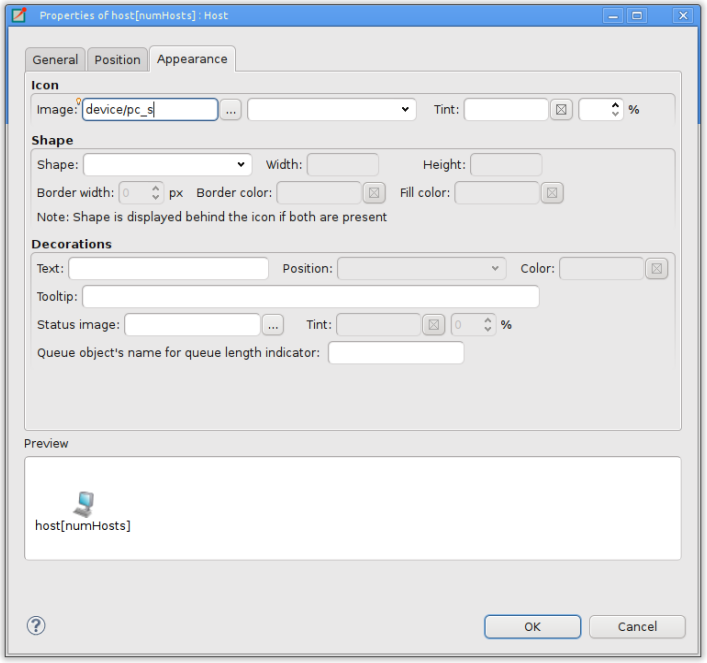


Fig. 2.4: Editing Visual Properties

Note: You can select multiple modules at the same time and open the *Properties* dialog to set their common properties simultaneously.

Changing a Module Parameter

To change a module parameter, right-click on it and select the *Parameters* menu item from the context menu. The dialog allows you to add or remove module parameters or assign values to them.

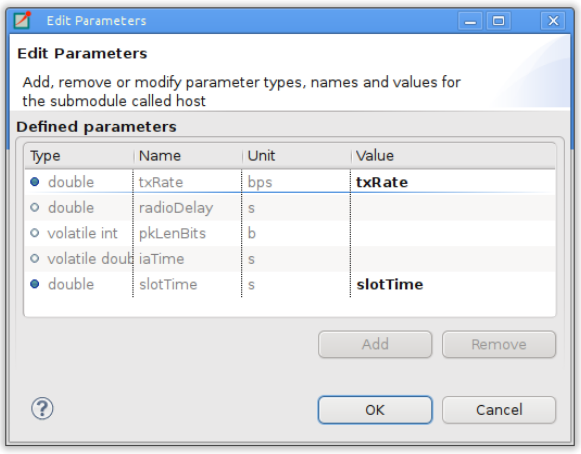


Fig. 2.5: Editing Module Parameters

Renaming Modules

To rename an existing module, select its context menu and choose *Rename* or click on an already selected module a second time. You can specify a new name for the module or even turn a submodule into a vector by adding `[vectorsize]` after its name. Alternatively, the name of a module can be set in the *Properties* dialog or can be edited by pressing `F6` when the module is selected.

Exporting a Module as an Image

A module can be exported using several image formats by selecting *Export Image* from the module's context menu.

Navigation

Double-clicking a submodule opens the corresponding module type in a NED editor. Selecting an element in the graphical editor and then switching to text mode places the cursor near the previously selected element in the text editor.

Navigating inside a longer NED file is easier if you open the *Outline View* to see the structure of the file. Selecting an element in the outline selects the same element in the graphical editor.

To see the selected element in a different view, select the element and right-click on it. Choose *Show In* from the context menu and select the desired view.

Opening a NED Type

If you only know the name of a module type or other NED element, you can use the *Open NED Type* dialog by pressing `Ctrl+Shift+N`. Type the name or search with wildcards. The requested type opens in an editor. This feature is not tied to the graphical editor: the *Open NED Type* dialog is available from anywhere in the IDE.

Setting Properties

Elements of the display string and other properties associated with the selected elements can be edited in the *Properties View*. The Property View is grouped and hierarchically organized; however, you can switch off this behavior on the view toolbar. Most properties can be edited directly in the *Properties View*, but some also have specific editors that can be activated by pressing the ellipsis button at the end of the field. Fields marked with a small light bulb support content assist. Use the `Ctrl+SPACE` key combination to get a list of possible values.

Note: The following functions are available only in source editing mode:

- Creating or modifying gates
- Creating grouped and conditional connections
- Adding or editing properties

2.4.2 Editing in Source Mode

The NED source editor supports all functionality expected from an Eclipse-based text editor, such as syntax highlighting, clipboard cut/copy/paste, unlimited undo/redo, folding, find/replace, and incremental search.

The NED source is continually parsed as you type, and errors and warnings are displayed as markers on the editor rulers. When the NED text is syntactically correct, the editor has full knowledge of "what is what" in the text buffer.

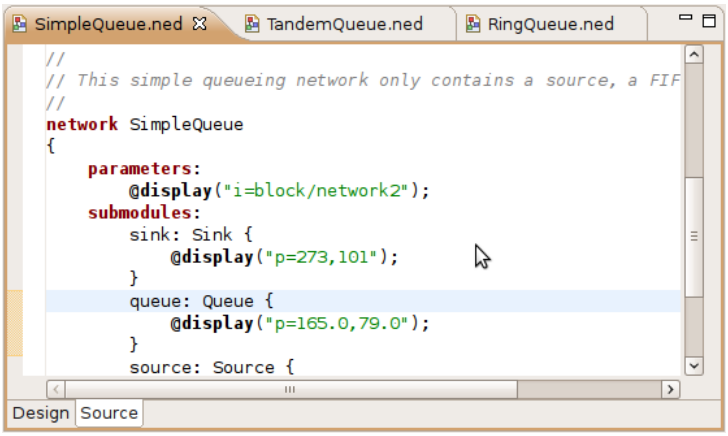


Fig. 2.6: NED Source Editor

Basic Functions

- Undo `Ctrl+Z`, Redo `Ctrl+Y`
- Indent/unindent code blocks `TAB` / `Shift+TAB`
- Correct indentation (NED syntax aware) `Ctrl+I`
- Find `Ctrl+F`, incremental search `Ctrl+J`
- Move lines `Alt+UP` `Alt+DOWN`

Tip: The following functions can help you explore the IDE:

- `Ctrl+Shift+L` pops up a window that lists all keyboard bindings, and
- `Ctrl+3` brings up a filtered list of all available commands.

Converting to the New NED Syntax

If you have an NED file with older syntax, you can still open it. A context menu item allows you to convert it to the new syntax. If the NED file already uses the new syntax, the *Convert to 4.x Format* menu item is disabled.

View Documentation

Hovering the mouse over a NED type name displays the documentation in a "tooltip" window, which can be made persistent by hitting `F2`.

Content Assist

If you need help, just press `Ctrl+SPACE`. The editor offers possible words or templates. This is context-sensitive, so it only offers valid suggestions. Content assist is also a good way to explore the new NED syntax and features.

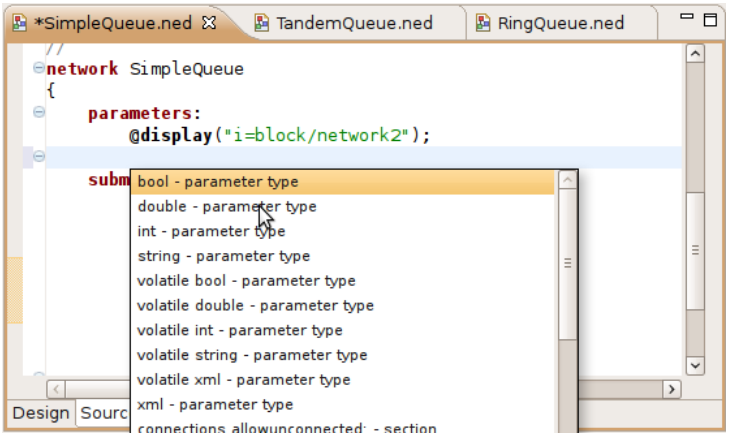


Fig. 2.7: NED Source Editor with content assist activated

Searching in NED Files

Selecting a text or moving the cursor over a word and pressing `Ctrl+Shift+G` searches for the selection in all NED files in the open projects. This function lets you quickly find references to the word or type currently under the cursor. The results are shown in the standard *Search View*.

Organizing Imports

Sometimes, it is very inconvenient to add the necessary import statements to the beginning of the NED file by hand. The IDE can do it for you (almost) automatically. Pressing `Ctrl+Shift+O` will make the IDE try to insert all necessary import statements. You will be prompted to specify the used packages in case of ambiguity.

Cleaning Up NED Files

This function does a general repair on all selected NED files by throwing out or adding import statements as needed, checking (and fixing) the file's package declaration, and reformatting the source code. It can be activated by clicking on the *Project* → *Clean Up NED Files* menu item from the main menu.

Commenting

To comment out the selected lines, press `Ctrl+/`. To remove the comment, press `Ctrl+/` again.

Formatting the Source Code

It is possible to reformat the whole NED file according to the recommended coding guidelines by activating the *Format Source* context menu item or by pressing the `Ctrl+Shift+F` key combination.

Note: Using the graphical editor and switching to source mode automatically re-formats the NED source code, as well.

Navigation

Holding the `Ctrl` key and clicking any identifier type will jump to the definition. Alternatively, move the cursor into the identifier and hit `F3` to achieve the same effect.

If you switch to graphical mode from text mode, the editor will try to locate the NED element under the cursor and select it in the graphical editor.

The Eclipse platform's bookmarking and navigation history facilities also work in the NED editor.

2.4.3 Other Features

Exporting Images

To export a compound module as a picture, select the compound module and bring up its context menu, select *Export Image* and choose file name and type. The module will be exported to the file system. BMP, PNG, JPEG, SVG and PDF formats are supported.

It is also possible to export images from all (or selected) NED files; the corresponding wizard can be found under *File* → *Export* in the menu.

2.5 Associated Views

There are several views related to the NED editor. These views can be displayed (if not already open) by choosing *Window* → *Show View* in the menu or by selecting a NED element in the graphical editor and selecting *Show In* from the context menu.

Note: If you are working with very large NED files, you may improve the performance of the editor by closing all NED file related views you do not need.

2.5.1 Outline View

The *Outline View* allows an overview of the current NED file. Clicking on an element will select the corresponding element in the text or graphical view. It has limited editing functionality; you can copy/cut/paste and delete an object.

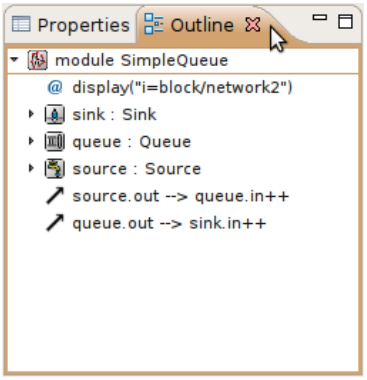


Fig. 2.8: Outline View

2.5.2 Property View

The *Property View* contains all properties of the selected graphical element. Visual appearance, name, type and other properties can be changed in this view. Some fields have specialized editors that can be activated by clicking on the ellipsis button in the field editor. Fields marked with a small light bulb icon have content assist support. Pressing `Ctrl+SPACE` will display the possible values the field can hold.

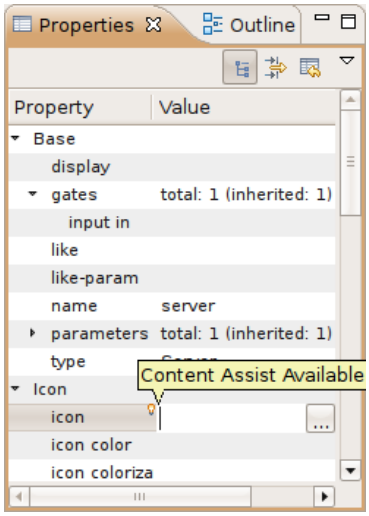


Fig. 2.9: Properties View

2.5.3 Palette View

The Palette is normally displayed on the left or right side of the editor area and contains tools to create various NED elements. It is possible to hide the Palette by clicking on the little arrow in the corner. You can also detach it from the editor and display it as a normal Eclipse View (*Window* → *Show View* → *Other* → *General* → *Palette*).

2.5.4 Problems View

The *Problems View* contains error and warning messages generated by the parser. Double-clicking a line will open the problematic file and move to the appropriate marker.

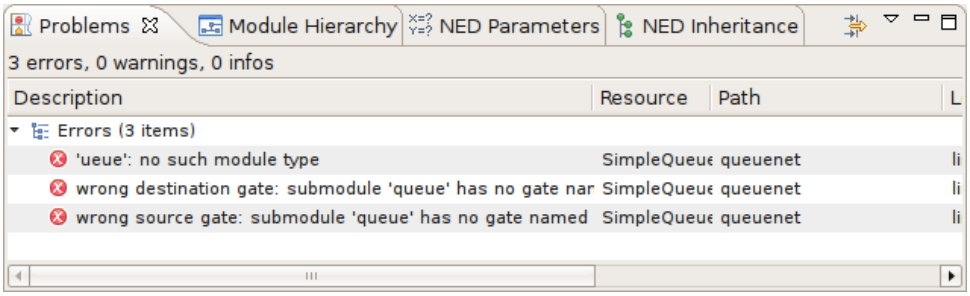


Fig. 2.10: Problems View

2.5.5 NED Inheritance View

The *Inheritance View* displays the relationship between different NED types. Select a NED element in the graphical editor or move the cursor into a NED definition and the *Inheritance View* will display the ancestors of this type. If you do not want the view to follow the selection in the editor, click the Pin icon on the view toolbar. This will fix the displayed type to the currently selected one.

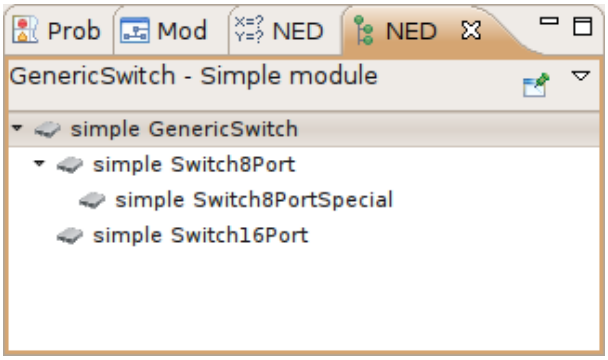


Fig. 2.11: NED Inheritance View

2.5.6 Module Hierarchy View

The *Module Hierarchy View* shows the contained submodules and their parameters, several levels deep. It also displays the parameters and other contained features.

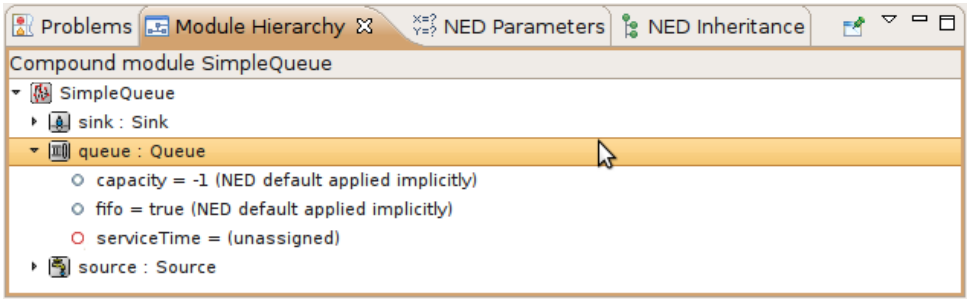


Fig. 2.12: Module Hierarchy View

2.5.7 Parameters View

The *Parameters View* shows the parameters of the selected module including inherited parameters.

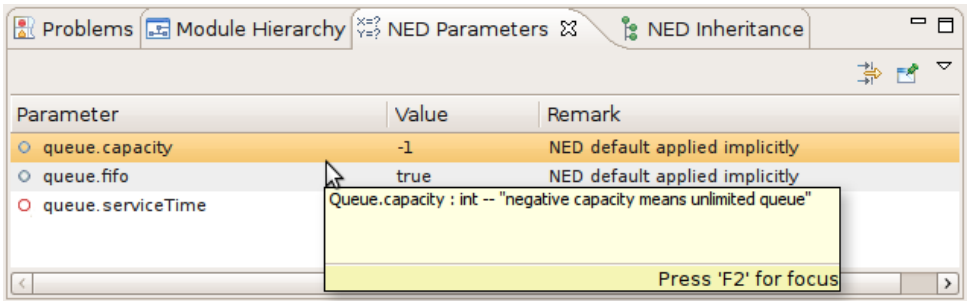


Fig. 2.13: Outline View

The latter two views are used mainly with the INI File Editor.

CHAPTER

THREE

EDITING INI FILES

3.1 Overview

In OMNEST, simulation models are parameterized and configured for execution using configuration files with the `.ini` extension, called INI files. INI files are text files, which can be edited using any text editor. However, OMNEST 4.x introduces a tool specifically designed for editing INI files. The INI File Editor is part of the OMNEST IDE and is highly effective in assisting the user to author INI files. It is a very useful feature because it has detailed knowledge of the simulation model, the INI file syntax, and the available configuration options.

Note: The syntax and features of INI files have changed since OMNEST 3.x. These changes are summarized in the "Configuring Simulations" chapter of the "OMNEST 4.x User Manual."

The INI File Editor is a dual-mode editor. The configuration can be edited using forms and dialogs, or as plain text. Forms are organized around topics such as general setup, Cmdenv, Qtenv, output files, extensions, and so on. The text editor provides syntax highlighting and auto completion. Several views can display information, which is useful when editing INI files. For example, you can see the errors in the current INI file or all the available module parameters in one view. You can easily navigate from the module parameters to their declaration in the NED file.

3.2 Creating INI Files

To create a new INI file, choose *File* → *New* → *Initialization File* from the menu. It opens a wizard where you can enter the name of the new file and select the name of the network to be configured.

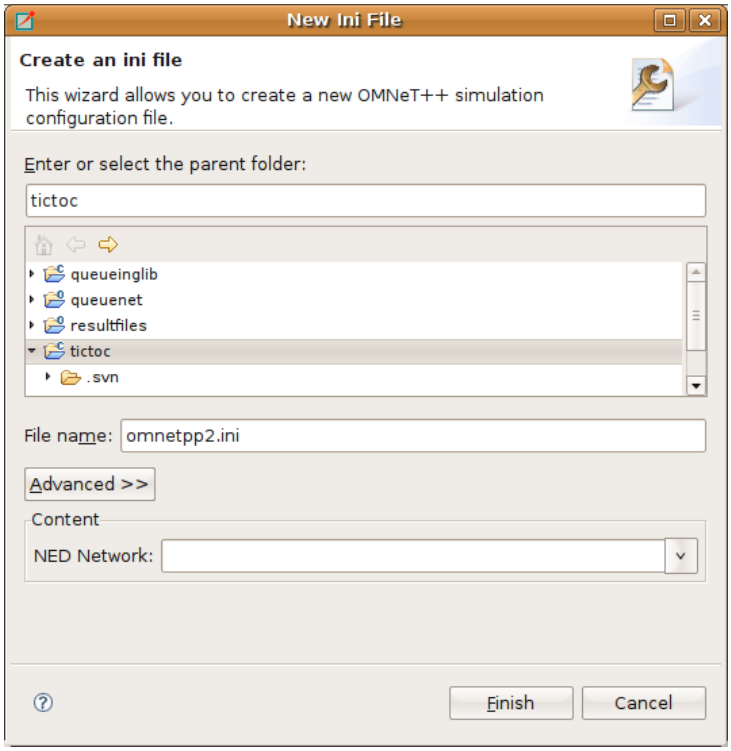


Fig. 3.1: New Initialization File dialog

3.3 Using the INI File Editor

The INI File Editor has two modes. The *Source* mode provides a text editor with syntax highlighting and auto completion of names. In the *Form* mode, you can edit the configuration by entering the values in a form. You can switch between the modes by selecting the tabs at the bottom of the editor.

3.3.1 Editing in Form Mode

The INI file contains the configuration of simulation runs. The content of the INI file is divided into sections. In the simplest case, all parameters are set in the General section. If you want to create several configurations in the same INI file, you can create named Configuration (Config) sections and refer to them with the `-c` option when starting the simulation. The Config sections inherit the settings from the General section or from other Config sections. This way, you can factor out the common settings into a "base" configuration.

On the first page of the form editor, you can edit the sections. The sections are displayed as a tree; the nodes inherit settings from their parents. The icon before the section name shows how many runs are configured in that section. You can use drag and drop to reorganize the sections. You can delete, edit, or add a new child to the selected section.

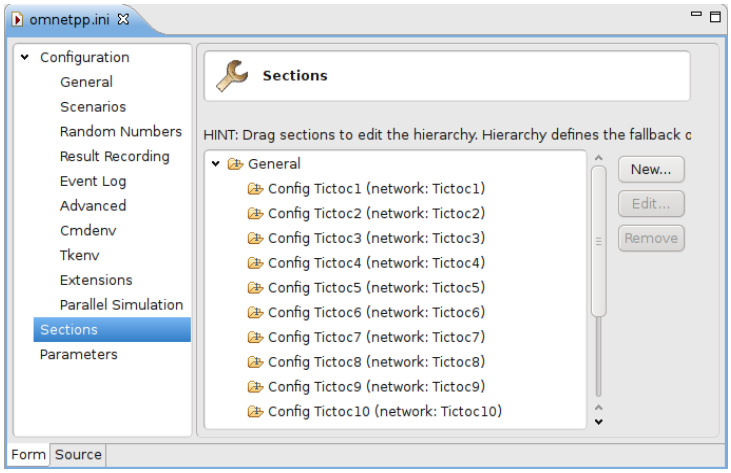


Fig. 3.2: Editing INI file sections

Table 3.1: Legend of Icons Before Sections

	contains a single run
	contains multiple replications (specified by 'repeat=. . . ')
	contains iteration variables
	contains multiple replications for each iteration

The Config sections have a name and an optional description. You can specify a fallback section other than General. If the network name is not inherited, it can be specified as well.

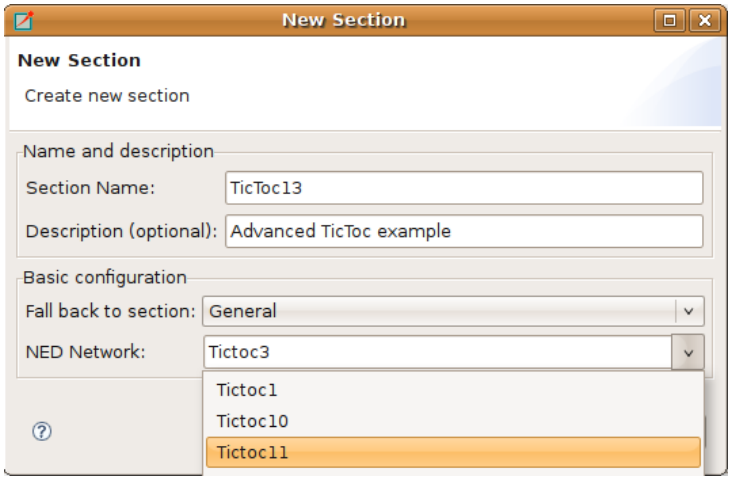


Fig. 3.3: Creating a new INI file section

On the *Parameters* page of the form editor, you can set module parameters. First, you have to select the section where the parameters are stored. After selecting the section from the list, the form shows the name of the edited network and the fallback section. The table below the list box shows the current settings of the section and all other sections from which it has inherited settings. You can move parameters by dragging them. If you click a table cell, you can edit the parameter name (or pattern), its value, and the comment attached to it. `Ctrl+SPACE` brings up a content assist. If you hover over a table row, the parameter is described in the tooltip that appears.

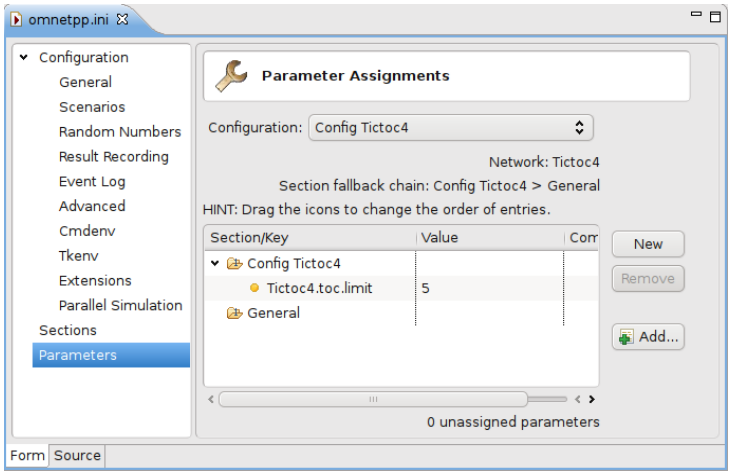


Fig. 3.4: Editing module parameters

New parameters can be added one by one by pressing the *New* button and filling the new table row. The selected parameters can be removed with the *Remove* button. If you press the *Add* button, you can add any missing parameters.

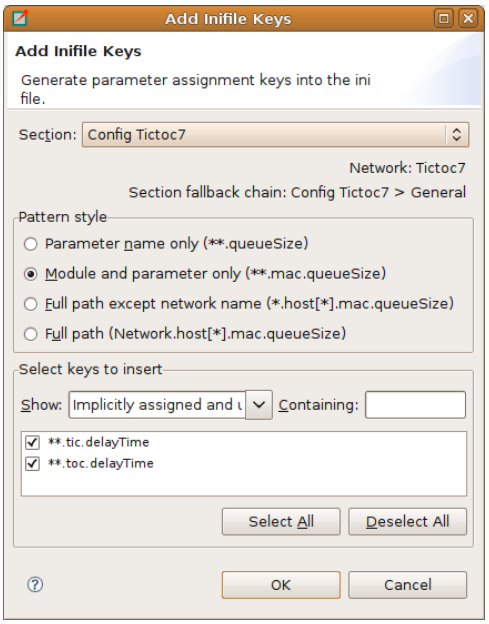


Fig. 3.5: Add missing module parameters dialog

The rest of the settings do not belong to modules (e.g., configuration of random number generators, output vectors, simulation time limit). These settings can be edited from the forms listed under the Configuration node. If the field has a default value and it is not set, the default value is displayed in gray. If its value is set, you can reset the default value by pressing the *Reset* button. These fields are usually set in the General section. If you want to specify them in a Config section, press the button and add a section-specific value to the opening table. If the table contains the Generic section only, then it can be collapsed again by pressing the button. Some fields can be specified in the General section only, so they do not have an button next to them.

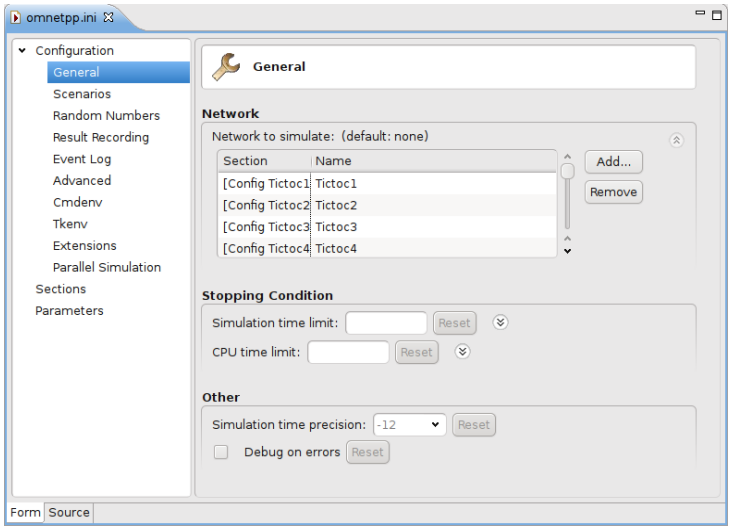


Fig. 3.6: Editing general configuration

3.3.2 Editing in Text Mode

If you want to edit the INI file as plain text, switch to the Source mode. The editor provides several features in addition to the usual text editor functions like copy/paste, undo/redo, and text search.

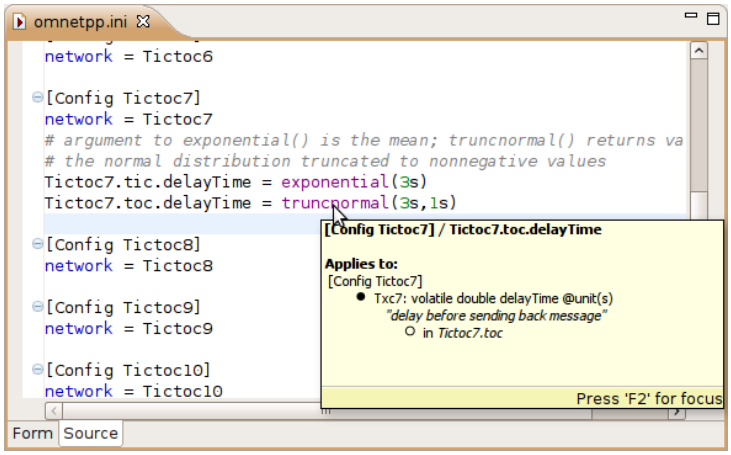


Fig. 3.7: Editing the INI file in text mode

Opening Old INI Files

When you open an INI file with the old format, the editor offers to convert it to the new format. It creates Config sections from Run sections and renames old parameters.

Content Assist

If you press `Ctrl+SPACE`, you will get a list of proposals valid at the insertion point. The list may contain section names, general options, and parameter names and values of the modules of the configured network.

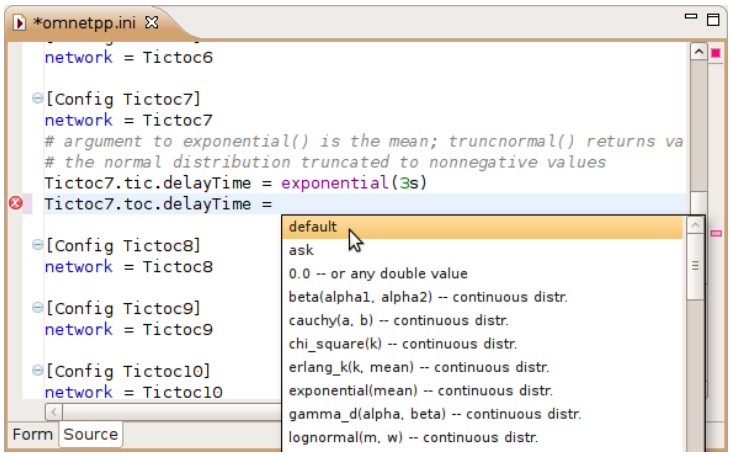


Fig. 3.8: Content assist in source mode

Tooltip

If you hover over a section or parameter, a tooltip appears showing the properties of the section or parameter. The tooltip for sections displays the inheritance chain, the network name, the number of errors and warnings, and the yet unassigned parameters. For parameters, the definition, description, and the module name are displayed.

Add Unassigned Parameters

You can add the names of unassigned module parameters to a Config section by choosing *Add Missing keys* from the context menu or pressing `Ctrl+Shift+O`.

Commenting

To comment out the selected lines, press `Ctrl+/`. To remove the comment, press `Ctrl+/` again.

Navigation

If you press the `Ctrl` key and click on a module parameter name, then the declaration of the parameter will be shown in the NED editor. You can navigate from a network name to its definition too.

Error Markers

Errors are marked on the left/right side of the editor. You can move to the next/previous error by pressing `Ctrl+.` and `Ctrl+,` respectively. You can get the error message in a tooltip if you hover over the  marker.

3.4 Associated Views

There are several views related to the INI editor. These views can be displayed (if not already open) by choosing the view from the *Window* → *Show View* submenu.

Note: If you are working with very large NED or INI files, you may improve the performance of the editor by closing all views related to INI files (Parameters, Module Hierarchy, and NED Inheritance View).

3.4.1 Outline View

The *Outline View* allows an overview of the sections in the current INI file. Clicking on a section will highlight the corresponding element in the text or form view.

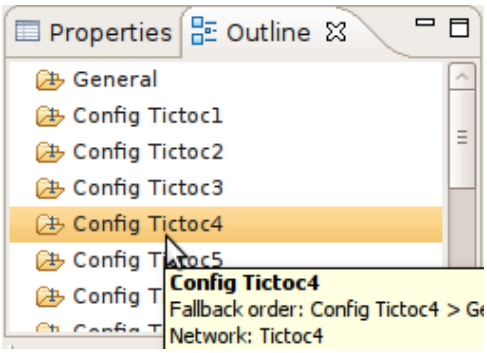


Fig. 3.9: Outline View showing the content of an INI file

3.4.2 Problems View

The *Problems View* contains error and warning messages generated by the parser. Double-clicking on a row will open the problematic file and move to the location of the problem.

3.4.3 Parameters View

The *Parameters View* shows parameters of the selected section including inherited parameters. It also displays the parameters that are unassigned in the configuration. When the toggle button on the toolbar is on, then all parameters are displayed; otherwise, only the unassigned ones are visible.

If you want to fix the content of the view, press the button. After pinning, the content of this view will not follow the selection made by the user in other editors or views.

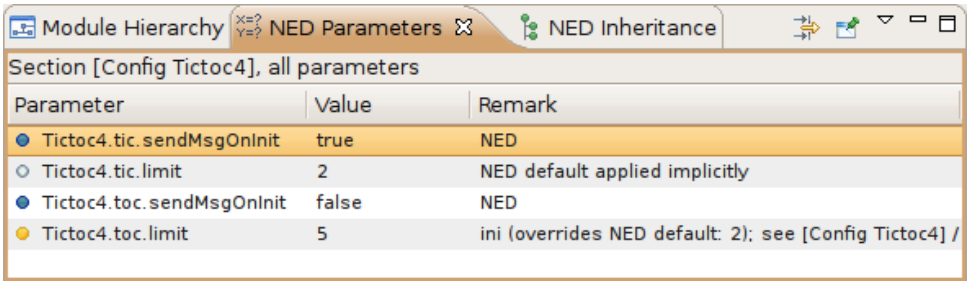


Fig. 3.10: Parameters View

Table 3.2: Legend of icons before module parameters

●	value is set in the NED file
◉	default from the NED file is explicitly set in the INI file (`**.paramname=default`)
○	default from the NED file is automatically applied because no value is specified in the INI file
●	value is set in the INI file (may override the value from the NED file)
◉	value is set in the INI file to the same value as the NED default
●	will ask the user at runtime (`**.paramname=ask`)
○	unassigned – has no values specified in the NED or INI files

Tip: Right-clicking on any line will show a context menu that allows you to navigate to the definition of that parameter or module.

3.4.4 Module Hierarchy View

The *Module Hierarchy View* shows the contained submodules, several levels deep. It also displays the module parameters and where their values come from (INI file, NED file, or unassigned).

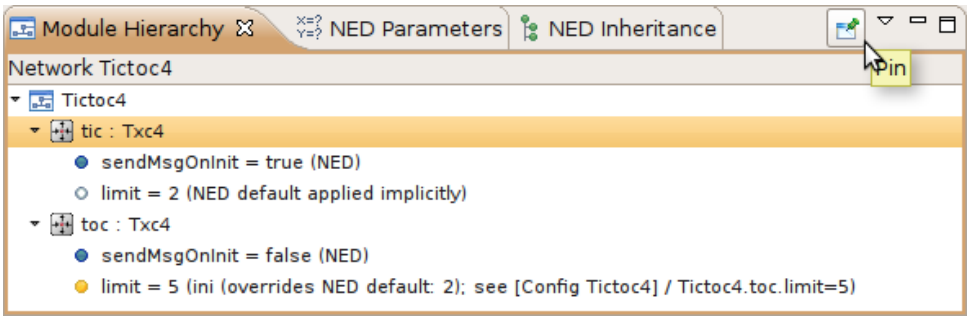


Fig. 3.11: Module Hierarchy View

Tip: Before you use the context menu to navigate to the NED definition, pin down the hierarchy view. This way, you will not lose the current context and content if the view will not follow the selection.

3.4.5 NED Inheritance View

The *NED Inheritance View* shows the inheritance tree of the network configured in the selected section.

CHAPTER

FOUR

EDITING MESSAGE FILES

4.1 Creating Message Files

Choosing *File* → *New* → *Message Definition (msg)* from the menu will bring up a wizard where you can specify the target directory and the file name for your message definition. You can choose to create an empty MSG file or choose from the predefined templates. Once you press the *Finish* button, a new MSG file will be created with the requested content.

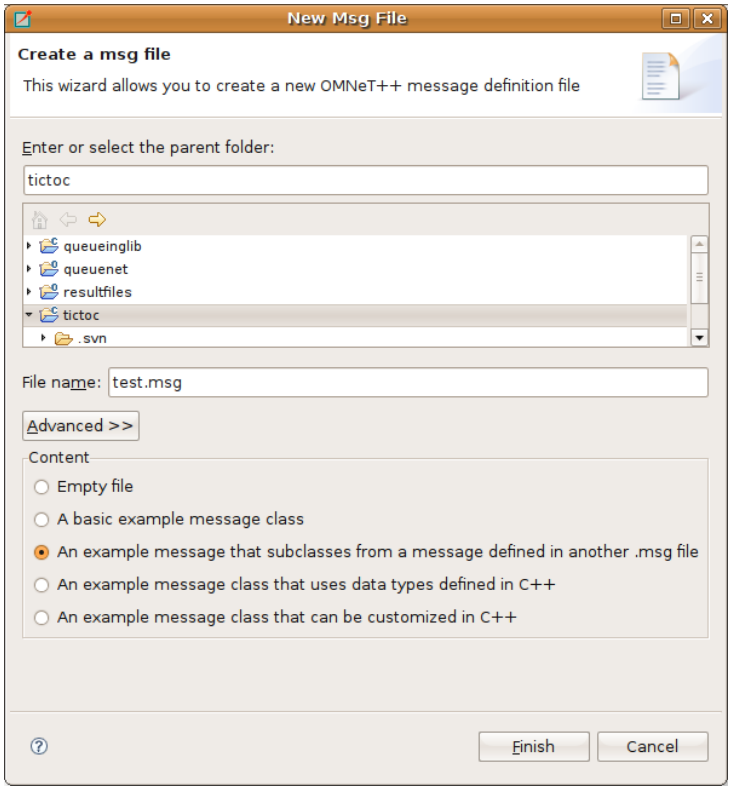


Fig. 4.1: Creating a new MSG file

4.2 The Message File Editor

The message file editor is a basic text editor with syntax highlight support.

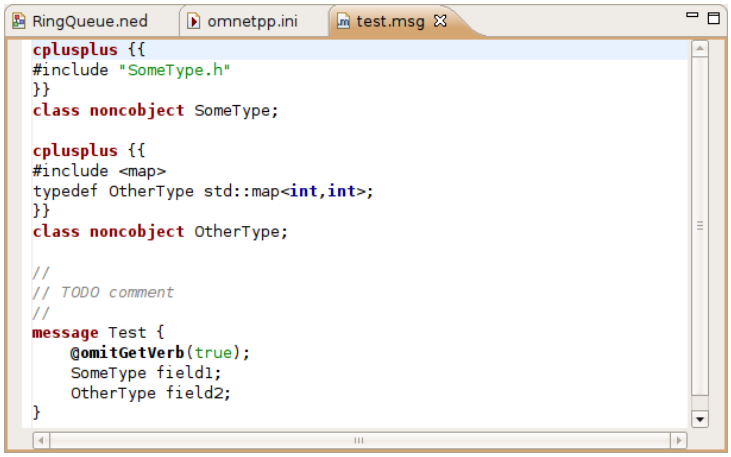


Fig. 4.2: Message File Editor

Note: Currently, the editor does not provide support for advanced features like content assistance or syntax-aware folding.

CHAPTER

FIVE

C++ DEVELOPMENT

5.1 Introduction

The OMNEST IDE contains editors, views, and other tools to assist you in developing your C++ code. C++ files open in the IDE in the C++ source editor. The C++ source editor supports syntax highlighting, documentation tooltips, content assist, automatic indentation, code formatting, refactoring, and several other useful features. The IDE also allows you to configure the build, start the build process, launch simulations, and debug the model without leaving the IDE.

Most of these features are provided by the Eclipse CDT (C/C++ Development Tooling) project (http://eclipse.org/cdt). This chapter briefly explains the basics of using CDT to develop simulation models. If you want to learn more about how to use CDT effectively, we recommend that you read the CDT documentation in the IDE help system (Help/Help Content).

The OMNEST IDE extends CDT with the following features to facilitate model development:

- A new OMNEST project creation wizard enables you to create simple, working simulation models in one step.
- Makefiles are automatically generated for your project based on the project build configuration. The built-in makefile generator is compatible with the command line **`opp_makemake`** tool and features deep makefiles, recursive make, cross-project references, invoking the message compiler, automatic linking with the OMNEST libraries, and support for building executables, shared libraries, or static libraries.
- Makefile generation and the project build system can be configured using a GUI interface.
- Project Features: Large projects can be partitioned into smaller units that can be independently excluded or included in the build. Disabling parts of the project can significantly reduce build time or make it possible to build the project at all.

5.2 Prerequisites

The OMNEST IDE (and the OMNEST simulation framework itself) requires a preinstalled compiler toolchain to function properly.

- On Windows: The OMNEST distribution comes with a preconfigured MinGW compiler toolchain. There is no need to manually install anything. By default, the IDE uses the Clang compiler from MinGW, but it is also possible to switch to the GCC compiler (also part of MinGW).
- On Linux: By default, the Clang compiler is used, but OMNEST falls back to using GCC if Clang is not present on the system. You have to install Clang or GCC on your system before trying to compile a simulation with OMNEST. Please read the Install Guide for detailed instructions.

- On macOS: You need to install Xcode Developer Tools to get compiler support before trying to compile a simulation with OMNEST. Please read the Install Guide for detailed instructions.

5.3 Creating a C++ Project

To create an OMNEST project that supports C++ development, select *File* → *New* → *OMNEST Project*.

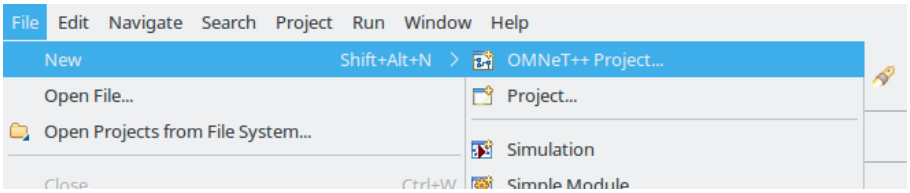


Fig. 5.1: Creating an OMNEST project

This menu item will bring up the *New OMNEST Project* wizard. The wizard lets you create an OMNEST-specific project, which includes support for NED, MSG, and INI file editing, as well as C++ development of simple modules.

On the first page of the wizard, specify the project name and ensure that the *Support C++ Development* checkbox is selected.

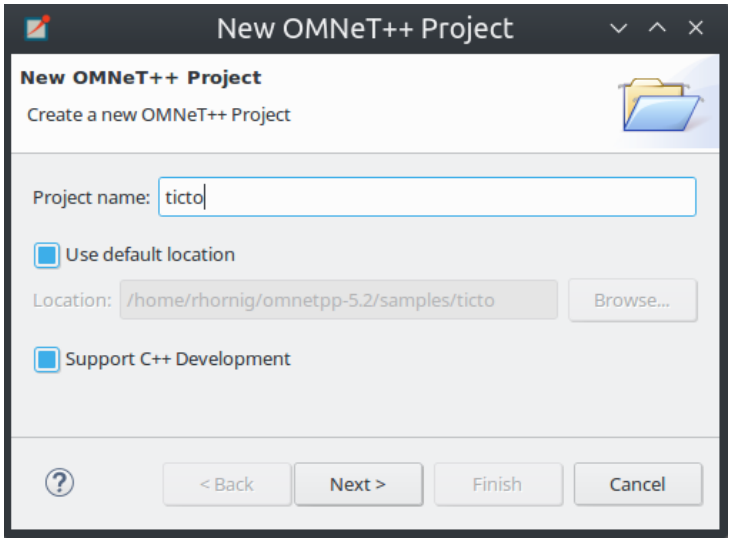


Fig. 5.2: Setting project name and enabling C++ support

Select a project template. A template defines the initial content and layout of the project.

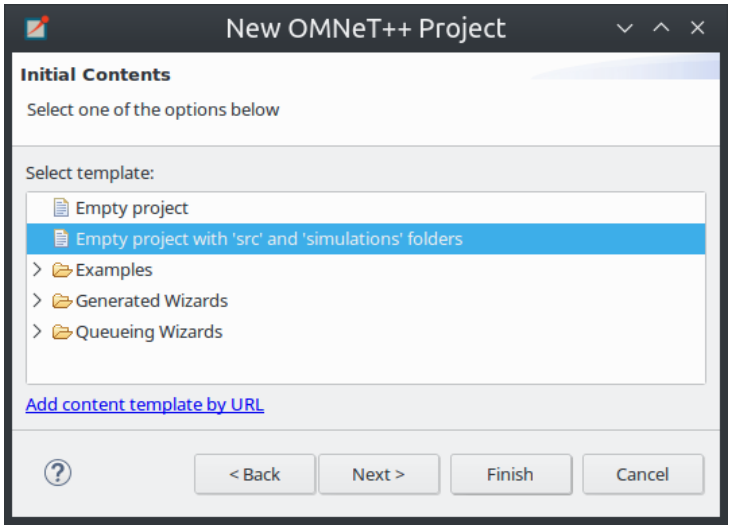


Fig. 5.3: Selecting a project template

Select a toolchain that is supported on your platform. Usually, you will see only a single supported toolchain, so there is no need to change anything on the page.

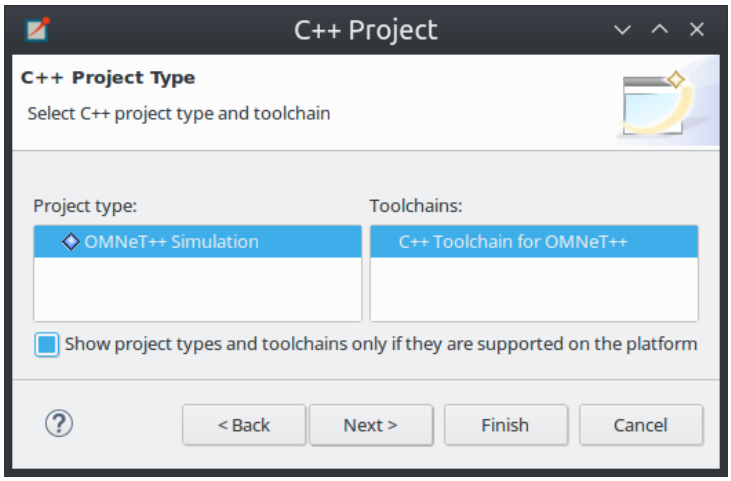


Fig. 5.4: Selecting a toolchain

Finally, select one or more from the preset build configurations. A configuration is a set of options that are associated with the build process. It is mainly used to build debug and release versions of your program.

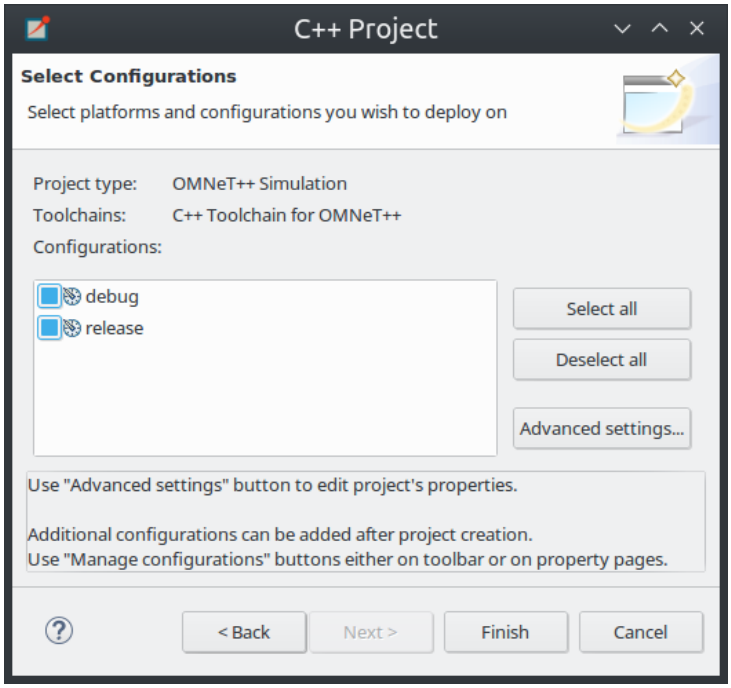


Fig. 5.5: Selecting configurations

Clicking the *Finish* button will create the project.

5.4 Editing C++ Code

The OMNEST IDE comes with a C/C++ editor. In addition to standard editing features, the C/C++ editor provides syntax highlighting, content assistance, and other C++ specific functionality. The source is continually parsed as you type, and errors and warnings are displayed as markers on the editor rulers.

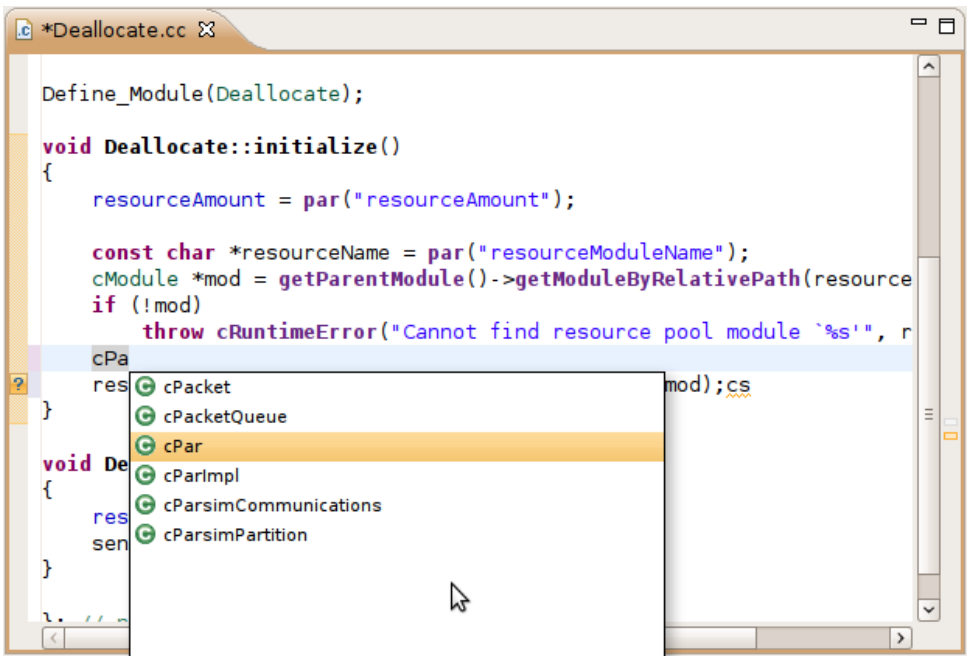


Fig. 5.6: C++ source editor

5.4.1 The C++ Editor

The C++ source editor provides the usual features of Eclipse-based text editors, such as syntax highlighting, clipboard cut/copy/paste, unlimited undo/redo, folding, find/replace, and incremental search.

The IDE scans and indexes the C++ files in your project in the background and provides navigation and code analysis features based on that knowledge. This database is kept up to date as you edit the source.

Basic Functions

Some of the most useful features of the source editor:

- Undo `Ctrl+Z`, Redo `Ctrl+Y`
- Switch between a C++ source and its matching header file `Ctrl+TAB`
- Indent/unindent code blocks `TAB` / `Shift+TAB`
- Correct indentation `Ctrl+I`
- Move lines `Alt+UP` / `Alt+DOWN`
- Find `Ctrl+F`, incremental search `Ctrl+J`

Tip: The following functions help you explore the IDE:

- `Ctrl+Shift+L` brings up a window listing all keyboard bindings, and
- `Ctrl+3` shows a filtered list of all available commands.

View Documentation

Hovering the mouse over an identifier displays its declaration and the documentation comment in a "tooltip" window. The window can be made persistent by hitting F2.

Tip: If you are on Ubuntu and see all-black tooltips, you need to change the tooltip colors in Ubuntu; see the Ubuntu chapter of the install-guide for details.

Content Assist

If you need help, just press `Ctrl+SPACE`. The editor will offer possible completions (variable names, type names, argument lists, etc.).

Navigation

Hitting `F3` or holding the `Ctrl` key and clicking an identifier will jump to the definition/declaration.

The Eclipse platform's bookmarking and navigation history facilities are also available in the C++ editor.

Commenting

To comment out the selected lines, press `Ctrl+/`. To remove the comment, press `Ctrl+/` again.

Open Type

Pressing Ctrl+Shift+T will bring up the *Open Element* dialog, which lets you type a class name, method name, or any other identifier and opens its declaration in a new editor.

Exploring the Code

The editor offers various ways to explore the code: Open Declaration `F3`, Open Type Hierarchy `F4`, Open Call Hierarchy `Ctrl+Alt+H`, Quick Outline `Ctrl+O`, Quick Type Hierarchy `Ctrl+T`, Explore Macro Expansion `Ctrl+=`, Search for References `Ctrl+Shift+G`, etc.

Refactoring

Several refactoring operations are available, such as Rename `Shift+Alt+R`.

Note: Several features such as content assist, go to definition, type hierarchy, and refactorings rely on the *Index*. The index contains the locations of all functions, classes, enums, defines, etc. in the project and referenced projects. Initial indexing of large projects may take a significant amount of time. The index is kept up to date mostly automatically, but occasionally it may be necessary to manually request reindexing the project. Index-related actions can be found in the *Index* submenu of the project's context menu.

5.4.2 Include Browser View

Dropping a C++ file into the *Include Browser View* displays the include files used by the C++ file (either directly or indirectly).

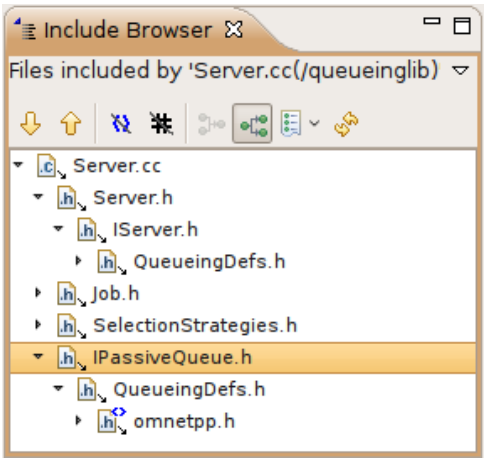


Fig. 5.7: Include Browser

5.4.3 Outline View

During source editing, the *Outline View* gives you an overview of the structure of your source file and can be used to quickly navigate within the file.

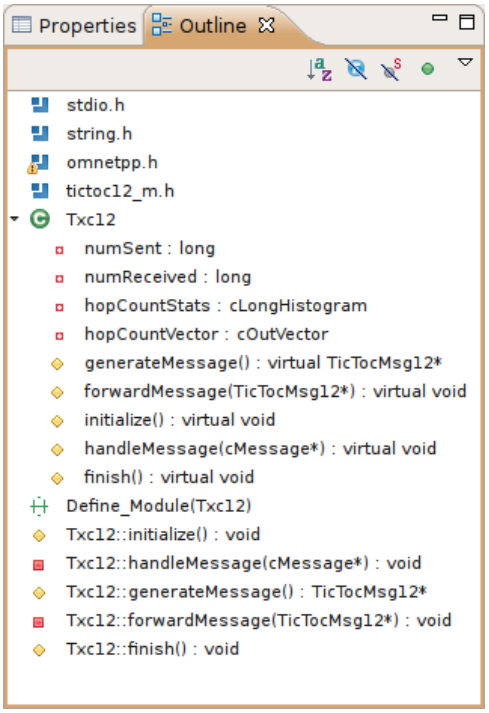


Fig. 5.8: Navigating with the Outline View

5.4.4 Type Hierarchy View

Displaying the C++ type hierarchy may be helpful in understanding the inheritance relationships among your classes (and among OMNEST classes).

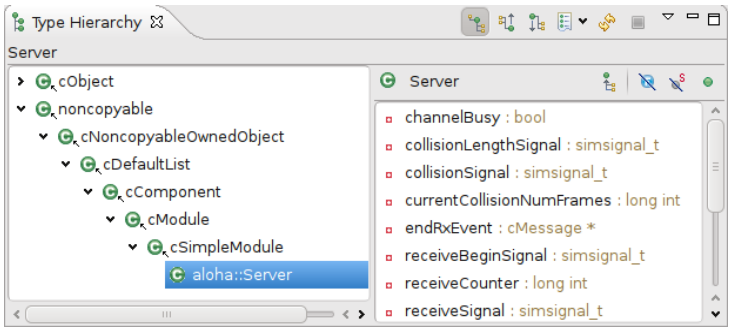


Fig. 5.9: C++ Type hierarchy

5.5 Building the Project

5.5.1 Basics

Once you have created your source files and configured your project settings, you can build the project by selecting *Build Project* from the *Project* menu or from the project context menu. You can also press `Ctrl+B` to build all open projects in the workspace.

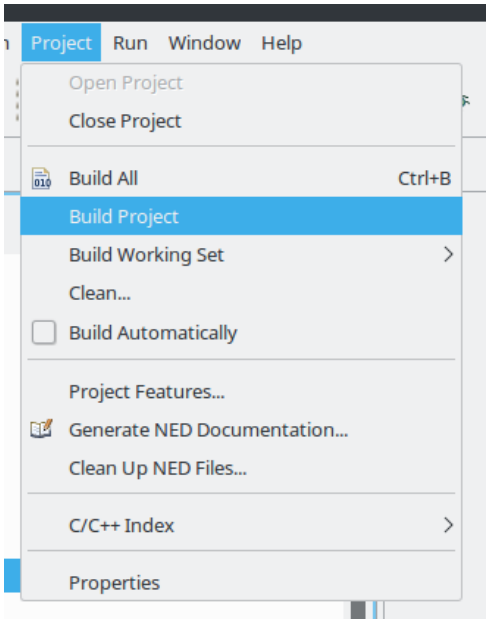


Fig. 5.10: Building a project

Build Output

The build output (standard output and standard error) is displayed in the *Console View* as the build progresses. Errors and warnings parsed from the output are displayed in the *Problems View*. Double-clicking a line in the *Problems View* will jump to the corresponding source line. Conversely, the *Console View* is more useful when you want to look at the build messages in their natural order (*Problems View* is usually sorted), for example when you get a lot of build errors and you want to begin by looking at the first one.

Makefile Generation

When you start the build process, a makefile is created or refreshed in each folder where makefile creation is configured. After that, make will be invoked with the `all` target in the folder configured as the build root.

Note: During the build process, the makefile will print out only the names of the compiled files. If you want to see the full command line used to compile each file, specify `V=1` (verbose on) on the make command line. To add this option, open *Project Properties* → *C/C++ Build* → *Behavior (tab)* and replace `all` with `all V=1` on the *Build* target line.

Cleaning the Project

To clean the project, choose *Clean* from the *Project* menu or *Clean Project* from the project context menu. This will invoke **`make`** with the `clean` target in the project's build root folder, and also in referenced projects. To clean only the local project and keep referenced projects intact, use *Clean Local* from the project context menu (see next section).

Referenced Projects and the Build Process

When you start the build, the IDE will build the referenced projects first. When you clean the project, the IDE will also clean the referenced projects first. This is often inconvenient (especially if your project depends on a large third-party project). To avoid cleaning the referenced projects, use *Clean Local* from the project context menu.

Build Configurations

A project is built using the active build configuration. A project may have several build configurations, where each configuration selects a compiler toolchain, debug or release mode, defines symbols, etc. To set the active build configuration, choose *Build Configurations* → *Set Active* from the project context menu.

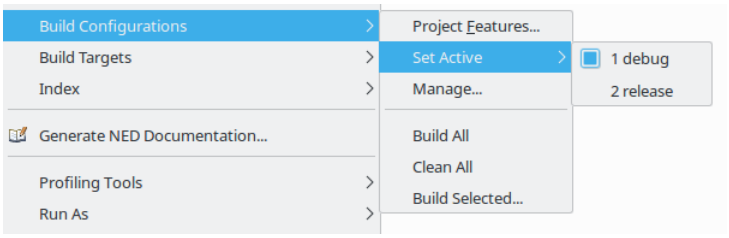


Fig. 5.11: Activating a build configuration

5.5.2 Console View

The *Console View* displays the output of the build process.

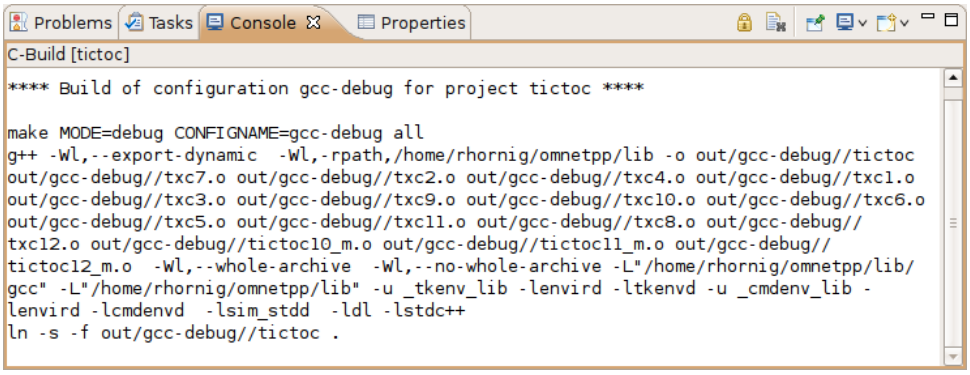


Fig. 5.12: Build output in a console

5.5.3 Problems View

The *Problems View* contains the errors and warnings generated by the build process. You can browse the problem list and double-click any message to go to the problem location in the source file. NED file and INI file problems are also reported in this view along with C++ problems. The editors are annotated with these markers as well. Hover over an error marker in the editor window to get the corresponding message as a tooltip.

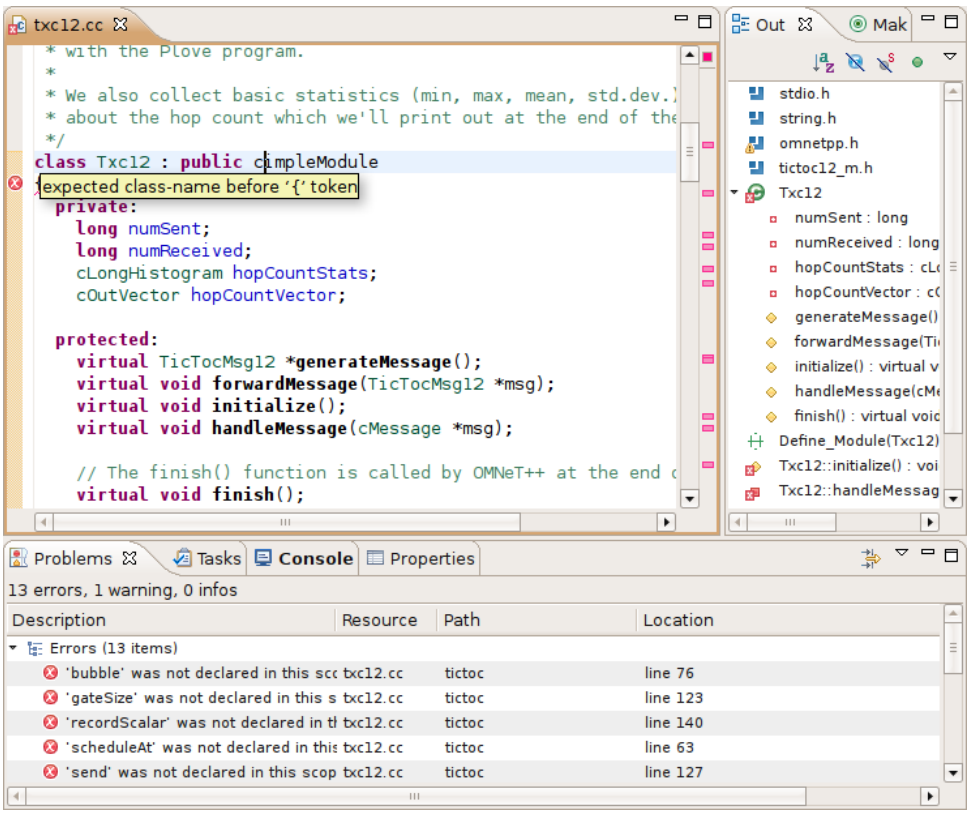


Fig. 5.13: C++ problems

5.6 Configuring the Project

5.6.1 Configuring the Build Process

The make invocation can be configured on the *C/C++ Build* page of the *Project Properties* dialog. Most settings are already set correctly and do not need to be changed. One exception is the *Enable parallel build* option on the *Behavior* tab that you may want to enable, especially if you have a multi-core computer.

> **Warning:** Do not set the number of parallel jobs to be significantly higher than the number of CPU cores you have. In particular, never turn on the *Use unlimited jobs* option, as it will start an excessive number of compile processes and can easily consume all available memory in the system.

We do not recommend changing any setting on property pages under the *C/C++ Build* tree node.

5.6.2 Managing Build Configurations

A project may have several build configurations, where each configuration describes the selected compiler toolchain, debug or release mode, extra include and linker paths, defined symbols, etc. You can activate, create, or delete build configurations under the *Build Configurations* submenu of the project context menu.

Note: Make sure that the names of all configurations contain the `debug` or `release` substring. The IDE launcher uses the name of the configuration to switch to the matching configuration depending on whether you want to debug or run the simulation.

5.6.3 Configuring the Project Build System

OMNEST uses makefiles to build the project. You can use a single makefile for the entire project or a hierarchy of makefiles. Each makefile may be hand-written (provided by you) or generated automatically. The IDE offers several options for automatically created makefiles.

The build system for an OMNEST project can be configured on the *OMNEST* → *Makemake* page of the *Project Properties* dialog. All settings on this page will affect all build configurations.

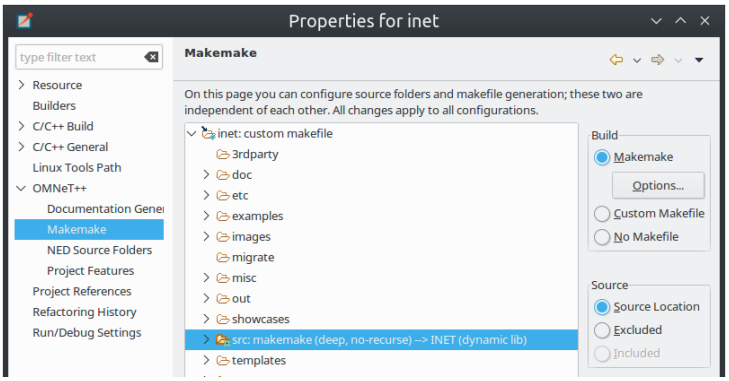


Fig. 5.14: Configuring Makefiles

Folders and Makefiles

The page displays the folder tree of the project. Using controls on the page (*Build* group in the top-right corner), you can declare that a selected folder contains a hand-written (custom) makefile or tell the IDE to generate a makefile for you. Generated makefiles will be automatically refreshed before each build. If a makefile is configured for a folder, the makefile kind will be indicated with a small decoration on the folder icon.

The build root folder is indicated with a small arrow. This is the folder in which the IDE's *Build* function will invoke the make command, so it should contain a makefile. It is expected that this makefile will build the entire project by invoking all other makefiles, either directly or indirectly. By default, the build root folder is the project root. This is usually fine, but if you really need to change the project build root, overwrite the *Build location* setting in the *C/C++ Build* page of the same dialog.

Note: All generated makefiles will be named `Makefile`. Custom makefiles are also expected to have this name.

Source Folders

In addition to makefiles, you also need to specify where your C++ files are located (source folders). This is usually the `src` folder of the project or, for small projects, the project root. It is also possible to exclude folders from a source folder. The controls on the bottom-right part of the dialog (*Source* group) allow you to set up source folders and exclusions for the project. Source files outside source folders or in an excluded folder will be ignored by both the IDE and the build process.

Note: Source folders and exclusions configured on this page actually modify the contents of the *Source Location* tab of the *C++ General* → *Paths and Symbols* page of the project properties dialog, and the changes will affect all build configurations.

Automatically created makefiles are by default *deep*, meaning that they include all (non-excluded) source files under them in the build. That is, a source file will be included in the build if it is under a source folder and covered by a makefile. (This applies to automatically generated makefiles; the IDE has no control over the behavior of custom makefiles.)

Makefile Generation

Makefile generation for the selected folder can be configured on the *Makemake Options* dialog, which can be accessed by clicking the *Options* button on the page. The dialog is described in the next section.

Command-line Build

To recreate your makefiles on the command line, you can export the settings by clicking the *Export* button. This action will create a file named `makemakefiles`. After exporting, execute **`make -f makemakefiles`** from the command line.

5.6.4 Configuring Makefile Generation for a Folder

Makefile generation for a folder can be configured on the *Makemake Options* dialog. To access the dialog, open the *OMNEST* → *Makemake* page in the *Project Properties* dialog, select the folder, make sure makefile generation is enabled for it, and click the *Options* button.

The following sections describe each page of the dialog.

The *Target* Tab

On the first, *Target* tab of the dialog, you can specify how the final target of the makefile is created.

- *Target type*: The build target can be an executable, a shared or static library, or the linking step may be omitted altogether. Makemake options: `--make-so`, `--make-lib`, `--nolink`
- *Export this shared/static library for other projects*: This option is observed if a library (shared or static) is selected as the target type and works in conjunction with the *Link with libraries exported from referenced projects* option on the *Link* tab. Namely, referencing projects will automatically link with this library if both the library is exported from this project AND linking with exported libraries is enabled in the referencing project. Makemake option: `--meta:export-library`

- *Target name*: You may set the target name. The default value is derived from the project name. Makemake option: `-o` (If you are building a debug configuration, the target name will be implicitly suffixed with the `_dbg` string.)
- *Output directory*: The output directory specifies where the object files and the final target will be created, relative to the project root. Makemake option: `-O`

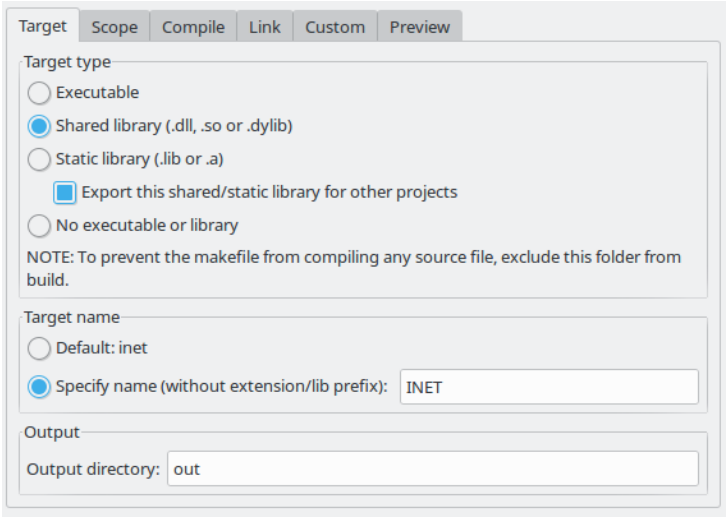


Fig. 5.15: Target definition

The *Scope* Tab

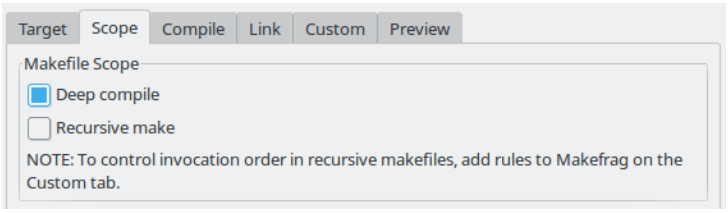


Fig. 5.16: Scope of the makefile

The *Scope* tab allows you to configure the scope of the makefile and specify which source files will be included.

- *Deep compile*: When enabled, the makefile will compile the source files in the entire subdirectory tree (except excluded folders and folders covered by other makefiles). When disabled, the makefile will only compile sources in the makefile's folder. Makemake option: `--deep`
- *Recursive make*: When enabled, the build will invoke make in all descendant folders that are configured to contain a makefile. Makemake option: `--meta:recurse` (expands to multiple `-d` options)
- *More » Additionally invoke make in the following directories*: If you want to invoke additional makefiles from this makefile, specify which directories should be visited (relative to this makefile). This option is useful if you want to invoke makefiles outside this source tree. Makemake option: `-d`

The *Compile* Tab

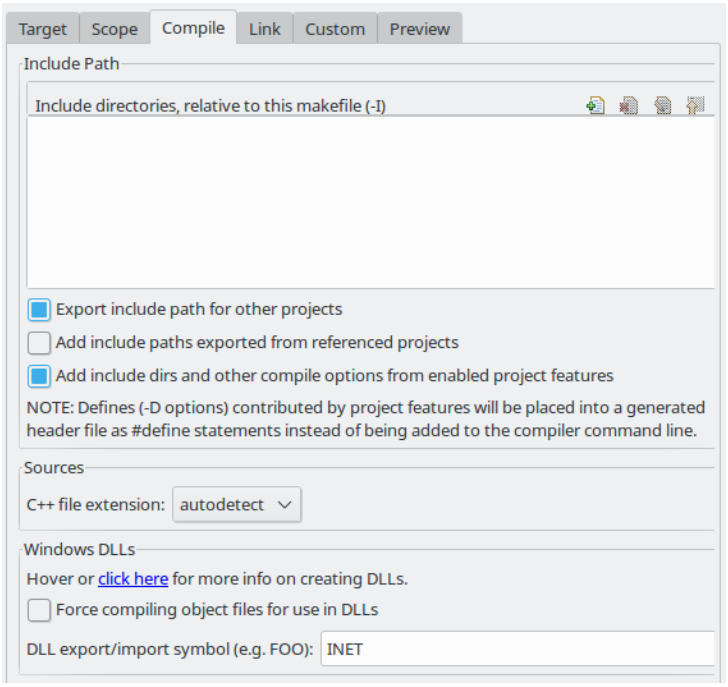


Fig. 5.17: Compiler options

The *Compile* tab allows you to adjust the parameters passed to the compiler during the build process.

Settings that affect the include path:

- *Export include path for other projects* makes this project's include path available for other dependent projects. This is usually required if your project expects other independent models to extend it in the future.
- *Add include paths exported from referenced projects* allows a dependent project to use header files from dependencies if those projects have exported their include path (i.e., the above option is turned on.)
- *Add include dirs and other compile options from enabled project features*: Project features may require additional include paths and defines to compile properly. Enabling this option will add those command-line arguments (specified in the `.oppfeatures` file) to the compiler command line.

Source files:

- *C++ file extension*: You can specify the source file extension used in the project (`.cc` or `.cpp`). We recommend using `.cc` in your projects. Makemake option: `-e`

If you build a Windows DLL, symbols you want to be available from other DLLs (or executables) need to be explicitly exported from the DLL. Functions, variables, and classes must be marked with `__declspec(dllexport)` when the DLL is compiled and with `__declspec(dllimport)` when you reference them from external code. This is achieved by defining a macro that expands differently in the two cases. The OMNEST convention is to name the macro `FOO_API`, where `FOO` is your project's short name. The macro should be defined as follows:

```
#if defined(FOO_EXPORT)
#  define FOO_API OPP_DLLEXPORT
```

(continues on next page)

(continued from previous page)

```
#elif defined(FOO_IMPORT)
#  define FOO_API OPP_DLLIMPORT
#else
#  define FOO_API
#endif
```

The above definition should be manually placed into a header file that is included by all headers where the macro is used. `OPP_DLLEXPORT` and `OPP_DLLIMPORT` are provided by `<omnetpp.h>`, and the generated makefile will provide the `FOO_EXPORT` / `FOO_IMPORT` macros that control the macro expansion via a compile option.

The `FOO_API` macro is used as illustrated in the following code:

```
class FOO_API ExportedClass {
   // public methods will be automatically exported
};
int FOO_API exportedFunction(...);
extern int FOO_API exportedGlobalVariable;
```

Settings for Windows DLLs:

- *Force compiling object files for use in DLLs*: If the makefile target is a DLL, OMNEST automatically compiles the sources for use in the DLL (defines the `FOO_EXPORT` macro, etc.), regardless of the state of this option. Rather, this option is useful if the makefile target is **not** a DLL, but the code compiled here will eventually end up in a DLL. Makemake option: `-S`
- *DLL export/import symbol*: Name for the DLL import/export symbol, i.e., `FOO` in the above examples. Makemake option: `-p`

The *Link* Tab

Link options allow you to fine-tune the linking steps at the end of the build process.

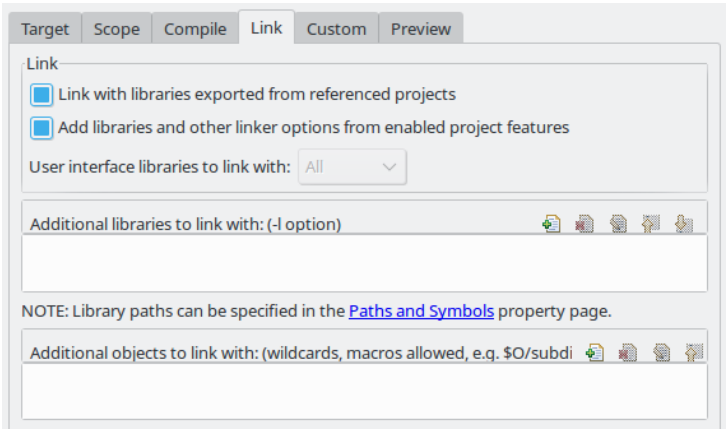


Fig. 5.18: Linker options

- *Link with libraries exported from referenced projects*: If your project references other projects that build static or dynamic libraries, you can instruct the linker to automatically link with those libraries by enabling this option. The libraries from the other projects must be exported via the *Export this shared/static library for other projects* option on the *Target* tab. Makemake option: `--meta:use-exported-libs`
- *Add libraries and other linker options from enabled project features*: Project features may require additional libraries and linker options to properly build. Enabling this option

will add those command line arguments (specified in the `.oppfeatures` file) to the linker command line.

- *User interface libraries to link with*: If the makefile target is an executable, you may specify which OMNEST user interface libraries (Cmdenv, Qtenv, or both) should be linked into the program. Makemake option: `-u`
- *More » Additional libraries to link with*: This box allows you to specify additional libraries to link with. Specify the library name without its path, possible prefix (`lib`), and file extension, and also without the `-l` option. The library must be on the linker path, which can be edited on the *Library Paths* tab of the *C/C++ General → Paths and Symbols* page of the *Project Properties* dialog. Makemake option: `-l`
- *More » Additional objects to link with*: Additional object files and libraries can be specified here. The files must be given with their full paths and file extensions. Wildcards and makefile macros are also accepted. Example: `$O/subdir/*.o`. Makemake option: none (files will become plain makemake arguments)

The *Custom* Tab

The *Custom* tab allows you to customize the makefiles by inserting handwritten makefile fragments into the automatically generated makefile. This lets you add additional targets, rules, variables, etc., to the generated makefile.

- *Makefrag*: If the folder contains a file named `makefrag`, its contents will be automatically copied into the generated makefile, just above the first target rule. `makefrag` allows you to customize the generated makefile to some extent. For example, you can add new targets (e.g., to generate documentation or run a test suite), new rules (e.g., to generate source files during the build), override the default target, add new dependencies to existing targets, or overwrite variables. The dialog lets you edit the contents of the `makefrag` file directly (it will be saved when you accept the dialog).
- *More » Fragment files to include*: Here, you can explicitly specify a list of makefile fragment files to include, instead of the default `makefrag`. Makemake option: `-i`

The *Preview* Tab

The *Preview* tab displays the command line options that will be passed to **`opp_makemake`** to generate the makefile. It consists of two parts:

- *Makemake options*: This is an editable list of makefile generation options. Most options map directly to checkboxes, edit fields, and other controls on the previous tabs of the dialog. If you check the *Deep compile* checkbox on the *Scope* tab, the `--deep` option will be added to the command line. If you delete `--deep` from the command line options, that will cause the *Deep compile* checkbox to be unchecked. Some options are directly understood by **`opp_makemake`**, others are "meta" options that the IDE will resolve to one or more **`opp_makemake`** options; see below.
- *Makemake options modified with CDT settings and with meta-options resolved*: This read-only text field is displayed for information purposes only. Not all options in the above options list are directly understood by **`opp_makemake`**; namely, the options that start with `--meta:` denote higher-level features offered by the IDE only. Meta options will be translated to **`opp_makemake`** options by the IDE. For example, `--meta:auto-include-path` will be resolved by the IDE to multiple `-I` options, one for each directory in the C++ source trees of the project. This field shows the **`opp_makemake`** options after the resolution of the meta options.

5.6.5 Project References and Makefile Generation

When your project references another project (such as the INET Framework), your project's build will be affected in the following way:

- Include path: Source folders in referenced projects will be automatically added to the include path of your makefile if the *Add include paths exported from referenced projects* option on the *Compile* tab is checked, and the referenced projects also enable the *Export include path for other projects* option.
- Linking: If the *Link with libraries exported from referenced projects* option on the *Link* tab is enabled, then the makefile target will be linked with those libraries in referenced projects that have the *Export this shared/static library for other projects* option checked on the *Target* tab.
- NED types: NED types defined in a referenced project are automatically available in referencing projects.

5.7 Project Features

5.7.1 Motivation

Long compile times are often an inconvenience when working with large OMNEST-based model frameworks like the INET Framework. The IDE feature called *Project Features* enables you to reduce build times by excluding or disabling parts of the model framework that you do not use for your simulation study. For example, when working on mobile ad-hoc simulations in INET, you can disable the compilation of Ethernet, IPv6/MIPv6, MPLS, and other unrelated protocol models. The word *feature* refers to a piece of the project codebase that can be turned off as a whole.

Additional benefits of project features include a less cluttered model palette in the NED editor, being able to exclude code that does not compile on your system, and enforcing cleaner separation of unrelated parts in the model framework.

Note: A similar effect could also be achieved by breaking up the model framework (e.g., INET) into several smaller projects, but that would cause other kinds of inconveniences for model developers and users alike.

5.7.2 What is a Project Feature

Features can be defined per project. As mentioned earlier, a feature is a portion of the project codebase that can be turned off as a whole, meaning it can be excluded from the C++ sources (and thus from the build) as well as from NED. Feature definitions are typically written and distributed by the project author, and end users are only presented with the option of enabling/disabling those features. A feature definition contains:

- `ID`, which is a unique identifier within the feature definition file.
- `Feature name`, for example `"UDP"` or `"Mobility examples"`.
- `Feature description`. This is a brief description of what the feature is or does, for example `"Implementation of the UDP protocol"`.
- `Labels`. This is a list of labels or keywords that facilitate grouping or finding features.
- `Initially enabled`. This is a boolean flag that determines the initial enablement of the feature.

- `Required features`. Some features may be built on top of others; for example, a HMIPv6 protocol implementation relies on MIPv6, which in turn relies on IPv6. Thus, HMIPv6 can only be enabled if MIPv6 and IPv6 are enabled as well. This is a space-separated list of feature IDs.
- `NED packages`. This is a space-separated list of NED package names that identify the code that implements the feature. When you disable the feature, NED types defined in those packages and their subpackages will be excluded; also, C++ code in the folders that correspond to the packages (i.e., in the same folders as excluded NED files) will also be excluded.
- `Extra C++ source folders`. If the feature contains C++ code that lives outside NED source folders (non-typical), those folders are listed here.
- `Compile options`, for example `-DWITH_IPv6`. When the feature is enabled, the compiler options listed here are either added to the compiler command line of all C++ files or they can be used to generate a header file containing all these defines so that header file can be included in all C++ files. A typical use of this field is defining symbols (`WITH_xxx`) that allows you to write conditional code that only compiles when a given feature is enabled. Currently, only the `-D` option (*define symbol*) is supported here.
- `Linker options`. When the feature is enabled, the linker options listed here are added to the linker command line. A typical use of this field is linking with additional libraries that the feature's code requires, for example libavcodec. Currently, only the `-l` option (*link with library*) is supported here.

5.7.3 The Project Features Dialog

Features can be viewed, enabled, and disabled on the *Project Features* page of the *Project Properties* dialog. The *Project* → *Project Features* menu item is a direct shortcut to this property page.

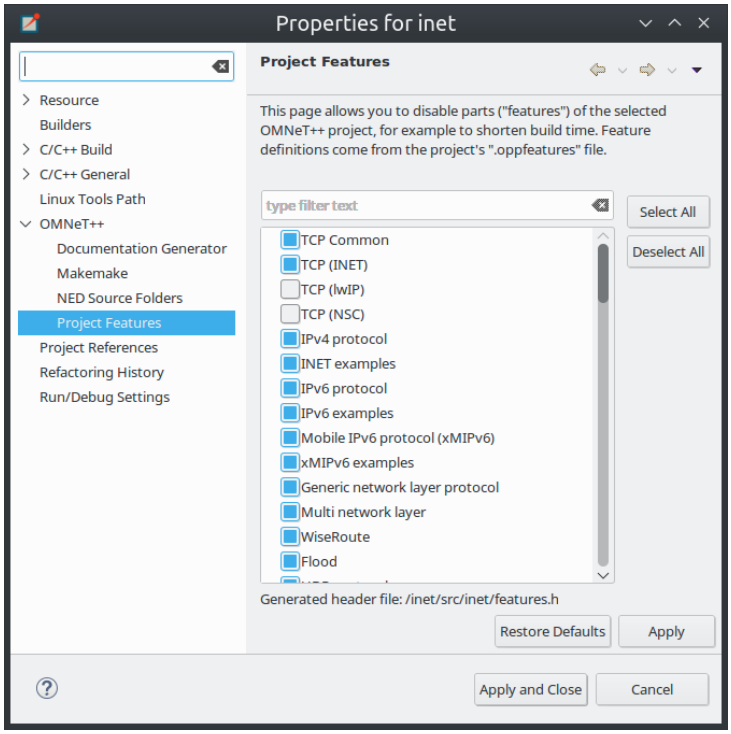


Fig. 5.19: The Project Features page

The central area of the dialog page lists the features defined for the project. Hovering the

mouse over a list item will display the description and other fields of the feature in a tooltip window. Checking an item enables the feature, and unchecking disables it.

When you enable a feature that requires other features to work, the dialog will ask for permission to enable the required features as well. Also, if you disable a feature that others depend on, they will be disabled too.

The *Apply*, *OK*, and *Cancel* buttons work as expected. *Restore Defaults* restores the features to their initial state (see the *Initially enabled* attribute above).

Above the list, there is a notification area in the dialog. If the IDE detects that your project's configuration is inconsistent with the feature enablements, it will display a warning there and offer a way to automatically fix the problems. Fixing means that the IDE will adjust the project's NED and C++ settings to make them consistent with the feature enablements. Such a check is also performed just before the build.

5.7.4 What Happens When You Enable/Disable a Feature

When you enable or disable a feature on the *Project Features* page, several project settings will be modified:

- NED package exclusions. This corresponds to the contents of the *Excluded package subtrees* list on the *NED Source Folders* property page. When a feature is disabled, its NED packages will be excluded (added to the list), and vice versa.
- C++ folder exclusions. This can be viewed/edited on the *Makemake* property page, and also on the *Source Location* tab of the *C/C++ General > Paths and Symbols* property page.
- Compile options. For example, if the feature defines preprocessor symbols (`-DWITH_xxx`), they can be used to generate a header file that contains the enabled macro definitions, and that file can be included in all C++ files.
- Linker options. For example, if the feature defines additional libraries to link with, they will be displayed on the *Libraries* tab of the *C/C++ General → Paths and Symbols* property page.

Note: Feature enablements are saved to the `.oppfeaturestate` file in the project root.

5.7.5 Using Features from the Command Line

Project Features can be easily configured from the IDE, but command line tools (**opp_makemake**, etc.) can also use them with the help of the **opp_featuretool** command.

If you want to build the project from the command line with the same feature combination the IDE is using, you need to generate the makefiles with the same **opp_makemake** options that the IDE uses in that feature combination. The **opp_featuretool makemakeargs** command (executed in the project's root directory) will show all the required arguments that you need to specify for the **opp_makemake** command to build the same output as the IDE. This allows you to keep the same features enabled no matter how you build your project.

Alternatively, you can choose *Export* on the *Makemake* page and copy/paste the options from the generated `makemakefiles` file. This method is not recommended because you must redo it manually each time after changing the enablement state of a feature.

5.7.6 The .oppfeatures File

Project features are defined in the .oppfeatures file in your project's root directory. This is an XML file, and it currently has to be written manually (there is no specialized editor for it).

The root element is <features>, and it may have several <feature> child elements, each defining a project feature. Attributes of the <features> element define the root(s) of the source folder(s) (cppSourceRoots) and the name of a generated header file that contains all the defines specified by the compilerFlags attribute in the enabled features. The fields of a feature are represented with XML attributes; attribute names are id, name, description, initiallyEnabled, requires, labels, nedPackages, extraSourceFolders, compileFlags, and linkerFlags. Items within attributes that represent lists (requires, labels, etc.) are separated by spaces.

Here is an example feature from the INET Framework:

```
<features cppSourceRoots="src" definesFile="src/inet/features.h">
    <feature
        id="TCP_common"
        name="TCP Common"
        description="The common part of TCP implementations"
        initiallyEnabled="true"
        requires=""
        labels=""
        nedPackages="
                        inet.transport.tcp_common
                        inet.applications.tcpapp
                        inet.util.headerserializers.tcp
                    "
        extraSourceFolders=""
        compileFlags="-DWITH_TCP_COMMON"
        linkerFlags=""
    />
```

5.7.7 How to Introduce a Project Feature

If you plan to introduce a project feature into your project, here's what you'll need to do:

- Isolate the code that implements the feature into a separate source directory (or several directories). This is because only whole folders can be declared as part of a feature; individual source files cannot.
- Check the remainder of the project. If you find source lines that reference code from the new feature, use conditional compilation (#ifdef WITH_YOURFEATURE) to make sure that the code compiles (and either works sensibly or throws an error) when the new feature is disabled. (Your feature should define the WITH_YOURFEATURE symbol, i.e. -DWITH_YOURFEATURE will need to be added to the feature compile flags.)
- Add the feature description into the .oppfeatures file of your project, including the required feature dependencies.
- Test. At the very least, test that your project compiles at all, both with the new feature enabled and disabled. More thorough, automated tests can be built using opp_featuretool.

5.8 Project Files

Eclipse, CDT, and the OMNEST IDE use several files in the project to store settings. These files are located in the project root directory and are normally hidden by the IDE in the *Project Explorer View*. The files include:

- `.project` : Eclipse stores the general project properties in this file, including project name, dependencies from other projects, and project type (i.e., whether OMNEST-specific features are supported or this is only a generic Eclipse project).
- `.cproject` : This file contains settings specific to C++ development, including the build configurations; and per-configuration settings such as source folder locations and exclusions, include paths, linker paths, symbols; the build command, error parsers, debugger settings, and so on.
- `.oppbuildspec` : Contains settings specific to OMNEST. This file stores per-folder makefile generation settings that can be configured on the *Makemake* page of the *Project Properties* dialog.
- `.oppfeatures` : Optionally contains the definitions of project features.
- `.oppfeaturestate` : Optionally contains the current enablement state of the features. (We do not recommend keeping this file under version control.)
- `.nedfolders` : Contains the names of NED source folders; this is the information that can be configured on the *NED Source Folders* page of the *Project Properties* dialog.
- `.nedexclusions` : Contains the names of excluded NED packages.

If you are creating a project where no C++ support is needed (i.e., you are using an existing precompiled simulation library and you only edit NED and Ini files), the `.cproject` and `.oppbuildspec` files will not be present in your project.

CHAPTER

SIX

LAUNCHING AND DEBUGGING

6.1 Introduction

The OMNEST IDE allows you to execute single simulations and simulation batches, as well as debug and, to some extent, profile simulations. You can choose whether you want the simulation to run in graphical mode (using *Qtenv*) or in console mode (using *Cmdenv*); which simulation configuration and run number to execute; whether to record an eventlog or not; and many other options.

When running simulation batches, you can specify the number of processes allowed to run in parallel, so you can take advantage of multiple processors or processor cores. The progress of the batch can be monitored, and you can also terminate processes from the batch if needed. Batches are based on the parameter study feature of INI files; you can read more about it in the OMNEST Simulation Manual.

Debugging support comes from the Eclipse C/C++ Development Toolkit (CDT), and beyond the basics (such as single-stepping, stack trace, breakpoints, watches, etc.), it also offers several conveniences and advanced functionality such as inspection tooltips, conditional breakpoints, and more. Debugging with CDT also has extensive literature on the Internet. Currently, CDT uses the GNU Debugger (gdb) as the underlying debugger.

Profiling support is based on the **`valgrind`** program, available at http://valgrind.org. Valgrind is a suite of tools for debugging and profiling on Linux. It can automatically detect various memory access and memory management bugs, and perform a detailed profiling of your program. Valgrind support is brought into the OMNEST IDE by the Linux Tools Project of Eclipse.

6.2 Launch Configurations

Eclipse, and therefore the IDE as well, uses *launch configurations* to store the details of the program to be launched: which program to run, the list of arguments and environment variables, and other options. Eclipse and its C/C++ Development Toolkit (CDT) already come with several types of launch configurations (e.g., "C/C++ Application"), and the IDE adds *OMNEST Simulation*. The same launch configuration can be used with the *Run*, *Debug*, and *Profile* buttons.

6.3 Running a Simulation

6.3.1 Quick Run

The simplest way to launch a simulation is by selecting a project, folder, INI, or NED file in *Project Explorer* and clicking the *Run* button on the toolbar. This will create a suitable launch configuration (if one does not already exist) and start the simulation. Alternatively, you can choose the *Run As* → *OMNEST Simulation* option from the context menu.

Details:

- If a folder is selected and it contains a single INI file, the IDE will use that file to start the simulation.
- If an INI file is selected, it will be used as the main INI file for the simulation.
- If a NED file is selected (containing a network definition), the IDE will search for INI files in the active projects and try to find a configuration that allows the network to start.

6.3.2 The Run Configurations Dialog

Launch configurations can be managed in the *Run Configurations* dialog. (Two other dialogs, *Debug Configurations* and *Profile Configurations*, are very similar and allow you to manage debugging/profiling aspects of launch configurations).

The *Run Configurations* dialog can be opened in various ways: via the main menu (*Run* → *Run Configurations*), via the context menu of a project, folder, or file (*Run As* → *Run Configurations*), via the green *Run* toolbar button (*Run Configurations* item in its attached menu, or by Ctrl-clicking any other menu item or the toolbar button itself).

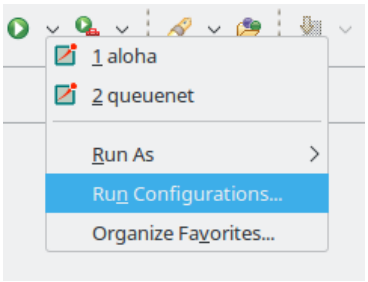


Fig. 6.1: One way to open the *Run Configurations* dialog

6.3.3 Creating a Launch Configuration

The OMNEST IDE adds a new Eclipse launch configuration type, *OMNEST Simulation*, that supports launching simulation executables. To create a new run configuration, open the *Run Configurations* dialog. In the dialog, select *OMNEST Simulation* from the tree and click the *New launch configuration* icon in the top-left corner. A blank launch configuration will be created, where you can give it a name at the top of the form.

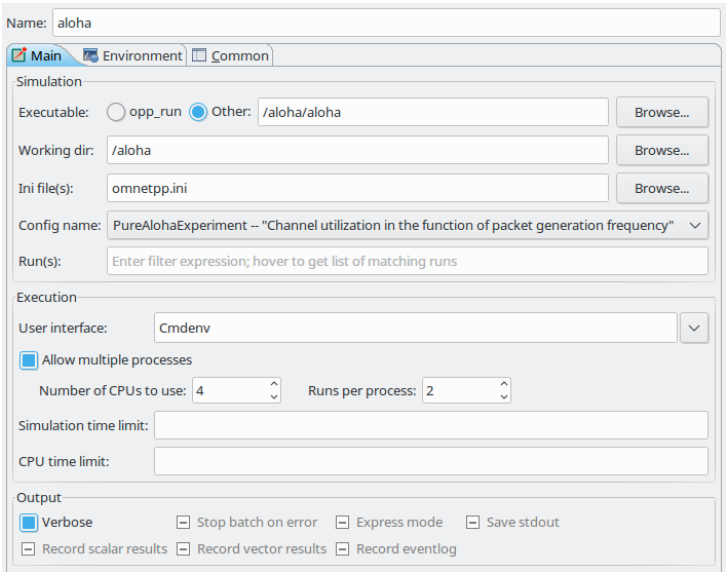


Fig. 6.2: The Simulation Launcher

The *Main* tab of the configuration dialog is designed to make the launching of simulations as easy as possible. The only required field is *Working directory*; all others have defaults. If you only select the working directory and the simulation program, it will start the first configuration from the `omnetpp.ini` file in the specified working directory.

Tip: Hover your mouse over the controls in this dialog to receive tooltip help for the selected control.

Note: The *Launch* dialog will try to figure out your initial settings automatically. If you select an INI file in the *Project Explorer View*, or the active editor contains an INI file before launching the *Run* dialog, the INI file and working directory fields will be automatically populated for you. The dialog will also try to guess the executable name based on the settings of your currently open projects.

- *Executable*: You must set the name of the simulation executable here. This is a workspace path. You may use the *Browse* button to select the executable directly. If your project output is a shared library, select *opp_run*. This will cause the IDE to use the **`opp_run`** or **`opp_run_dbg`** helper executable with the `-l` option to run the simulation. Make sure that the *Dynamic Libraries* field in the advanced section contains the libraries you want to load.
- *Working directory*: Specifies the working directory of the simulation program. This is a workspace path. Note that values in several other fields in the dialog are treated as relative to this directory, so changing the working directory may invalidate or change the meaning of previously selected entries in other fields of the dialog.
- *Initialization file(s)*: You should specify one or more INI files that will be used to launch

the simulation. The default is `omnetpp.ini`. Specifying more than one file (separated by space) will cause the simulation to load all those files in the specified order.

- *Config name*: Once you specify a valid INI file, this box will display all the Config sections in that file. In addition, it will display the description of each section and the information regarding which Config section is extended by it. You can select which Configuration to launch.

Note: The working directory and the INI file must contain valid entries before setting this option.

- *Runs*: You can specify which run(s) to execute for the simulation. An empty field corresponds to all runs. You can specify run numbers or a filter expression that refers to iteration variables. Use commas and the `..` operator to separate the run numbers; for example, `1,2,5..9,20` corresponds to run numbers 1,2,5,6,7,8,9,20. You can also specify run filters, which are boolean expressions involving constants and iteration variables (e.g., `$numHosts>5 and $numHosts<10`). Running several simulations in this manner is called batch execution.

Tip: If the executable name and the INI file have already been selected, hover the mouse over the field to get the list of matching runs.

- *User interface*: You can specify which UI environment should be used during execution. The dialog offers *Cmdenv* (command-line UI) and *Qtenv* (Qt-based GUI). If you have a custom user interface, you can also specify its name here. Make sure that the code of the chosen UI library is available (linked into the executable/library or loaded dynamically).

Note: Batch execution and progress feedback during simulation are only supported when using *Cmdenv*.

- ***Allow multiple processes*: With batch execution, you can tell the launcher to keep two or more**
  simulations running at a time or to start a new simulation process after a certain number of runs have been executed. This way, you can take advantage of multiple CPUs or CPU cores. You can set the number of CPUs to use and the number of runs to execute in a single process.

  > **Warning:** Only use this option if your simulation is CPU-limited and you have enough physical RAM to support all of the processes at the same time. Do not set it higher than the number of physical processors or cores you have in your machine.

- *Simulation time limit* and *CPU time limit* can also be set to limit the runtime length of the simulation from the launch dialog, in case those were not set in the INI file.
- *Output options*: Various options can be set regarding simulation output. These checkboxes may be in one of three states: checked (on), unchecked (off), and grayed out (unspecified). When a checkbox is grayed out, the launcher lets the corresponding configuration option from the INI file take effect.
- Clicking on the *More* >>> link will reveal additional controls.
- *Dynamic libraries*: A simulation may load additional DLLs or shared libraries before execution, or your entire simulation may be built as a shared library. The *Browse* button is available to select one or more files (use `Ctrl` + click for multiple selection). This option can be used to load simulation code (i.e., simple modules), user interface

libraries, or other extension libraries (scheduler, output file managers, etc.). The special macro `${opp_shared_libs:/workingdir}` expands to all shared libraries provided by the current project or any other project on which you currently depend.

Note: If your simulation is built as a shared library, you must use the **opp_run** stub executable to start it. **opp_run** is basically an empty OMNEST executable that understands all command line options but does not contain any simulation code.

Warning: If you use external shared libraries (i.e., libraries other than the ones provided by the current open projects or OMNEST itself), ensure that the executable part has access to the shared library. On Windows, you must set the `PATH` environment variable, while on Linux and Mac, you must set the `LD_LIBRARY_PATH` environment variable to point to the directory where the DLLs or shared libraries are located. You can set these variables either globally or in the *Environment* tab of the *Launcher Configuration Dialog*.

- *NED Source Path*: The directory or directories where the NED files are read from.

Tip: The variable `${opp_ned_path:/workingdir}` refers to an automatically computed path (derived from project settings). If you want to add additional NED folders to the automatically calculated list, use the `${opp_ned_path:/workingdir}:/my/additional/path` syntax.

- *Image path*: A path that is used to load images and icons in the model.
- *Additional arguments*: Other command-line arguments can be specified here and will be passed to the simulation process.
- *Build before launch*: This section allows you to configure the behavior of automatic build before launching. The build scope can be set to *this project and all its dependencies*, *this project only*, or you can turn off autobuild before launch. The active configuration switching during the build can also be configured here (*Ask*, *Switch*, *Never switch*).

Related Command-Line Arguments

Most settings in the dialog correspond to command-line options for the simulation executable. Here is a summary:

- Initialization files: maps to multiple `-f <inifile>` options.
- Configuration name: adds a `-c <configname>` option.
- Run number: adds a `-r <runnumber/filter>` option.
- User interface: adds a `-u <userinterface>` option.
- Dynamically loaded libraries: maps to multiple `-l <library>` options.
- NED source path: adds a `-n <nedpath>` option.

6.3.4 Debug vs. Release Launch

The launcher automatically determines whether to start the release or debug build of the model. When running, release-mode binaries are used automatically. For debugging, debug builds are started (i.e., those with a binary name ending with the `_dbg` suffix). Before starting the simulation, the launcher checks if the binary is up to date and triggers a build process (and also changes the active configuration) if necessary.

6.4 Batch Execution

OMNEST INI files allow you to run a simulation several times with different parameters. You can specify loops or conditions for specific parameters.

```
[Config PureAlohaExperiment]
description = "Channel utilization in the function of packet generation frequency"
repeat = 2
sim-time-limit = 300min
**.vector-recording = false
Aloha.host[*].iaTime = exponential(${mean=1,1.5,2,3,4,5..21 step 2}s)
```

Fig. 6.3: Iteration variable in the INI file

Note: Batch running is supported only in the command-line environment.

If you create an INI file configuration (a [Config] section) with one or more iteration variables, you will be able to run your simulations and explore the parameter space defined by those variables. Basically, the IDE creates the Cartesian product from these variables and assigns a run number to each combination. To execute one, several, or all runs of the simulation, you can specify the *Run number* field in the *Run Dialog*. You can specify a single number (e.g., `3`), a combination of several numbers (e.g., `2,3,6,7..11`), all run numbers (using `*`), or a boolean expression using constants and iteration variables (e.g., `$numHosts>5 and $numHosts<10`).

Tip: If you have already specified your executable, selected the configuration to be run, and chosen the command line environment, you can hover over the *Run number* field. This will give you a description of the possible runs and how they are associated with the iteration variable values (the tooltip is calculated by executing the simulation program with the `-x Configuration -G` options in command line mode).

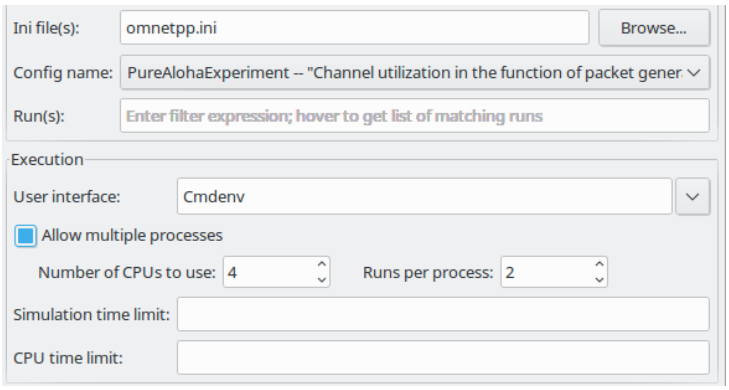


Fig. 6.4: Iteration loop expansion in a tooltip

If you have a multi-core or multi-processor system and sufficient memory, you can set the *Processes to run parallel* field to a higher number. This will allow the IDE to start more simulation processes in parallel, resulting in a much lower overall simulation time for the whole batch.

Warning: Be aware that you need enough memory to run all these processes in parallel. We recommend using this feature only if your simulation is CPU-bound. If you do not have enough memory, your operating system may start using virtual memory, dramatically decreasing the overall performance.

6.5 Debugging a Simulation

The OMNEST IDE integrates with the CDT (C/C++ Development Tooling) of Eclipse, which includes debugging support as well. The CDT debugger UI relies on **gdb** to do the actual work.

6.5.1 Starting a Debug Session

Launching a simulation in debug mode is very similar to running it (see previous sections), except you need to select the *Debug* toolbar icon or menu item instead of *Run*. The same launch configurations used for running are used for debugging, so if you open the *Debug Configurations* dialog, you will see the same launch configurations as in the *Run* dialog. The launcher automatically uses the debug build of the model (i.e., the executable with a `_dbg` suffix). The dialog will have extra tab pages where you can configure the debugger and other details.

Note: If you have problems starting the debug session, check the following:

- Ensure that your executable is built with debug information.
- Verify that you can run the same executable without any issues (using the same launch configuration, but with the addition of a `_dbg` suffix to the executable name).
- Make sure the debugger type is set correctly on the *Debugger* tab of the *Launch* dialog.

Warning: Batch (and parallel) execution is not possible in this launch type, so you can only specify a single run number.

6.5.2 Using the Debugger

The CDT debugger provides functionality that can be expected from a good C/C++ debugger: run control (run, suspend, step into, step over, return from function, drop to stack frame), breakpoints (including conditional and counting breakpoints), watchpoints (expression breakpoints that stop the execution whenever the value of a given expression changes), and watching and inspecting variables. Access to machine-level details such as disassembly, registers, and memory is also available.

Source code is shown in the editor area, and additional information and controls are displayed in various Views such as *Debug*, *Breakpoints*, *Expressions*, *Variables*, *Registers*, *Memory*, etc.

CDT's conversation with gdb can also be viewed in the appropriate pages of the *Console View*. (Click the *Display Selected Console* icon and choose either *gdb* or *gdb traces* from the menu.)

Tip: If you have a pointer in the program that points to an array (of objects, etc.), you can have it displayed as an array too. In *Variables*, right-click the variable and choose *Display As Array* from the menu. You will be prompted for a start index and the number of elements to display.

More information on the debugger is available in the CDT documentation, which is part of the IDE's Help system. See the *C/C++ Development User Guide*, chapter *Running and debugging projects*.

6.5.3 Pretty Printers

Many programs contain data structures whose contents are difficult to comprehend by looking at "raw" variables in the program. One example is the `std::map<T>` class, which is essentially a dictionary but implemented with a binary tree, making it practically impossible to figure out with a C++ debugger what data a concrete map instance contains.

The solution gdb offers to this problem is pretty printers. Pretty printers are Python classes that gdb invokes to transform some actual data structure into something that is easier for humans to understand. The `*.py` files that provide and register these pretty printers are usually loaded via gdb's startup script, `.gdbinit` (or some `.gdbinit.py` script, as gdb allows startup scripts to be written in Python too).

The OMNEST IDE includes pretty printers for container classes in the standard C++ library (such as `std::map<T>` and `std::vector<T>`), as well as for certain OMNEST data types, such as `simtime_t`. These scripts are located under `misc/gdb/` in the OMNEST root directory. The IDE also supports project-specific pretty printers: if the root folder of the debugged project contains a `.gdbinit.py` file, it will be loaded by gdb. (The project's `.gdbinit.py` file can then load further Python scripts, such as from an `etc/gdb/` folder of the project.)

Pretty printer loading works as follows: the IDE invokes gdb with `misc/gdb/gdbinit.py` as the startup script (for new launch configurations, the *GDB command file* field on the *Debugger* tab is set to `${opp_root}/misc/gdb/gdbinit.py`). This script loads the pretty printers under `misc/gdb`, as well as the project-specific pretty printers.

Tip: If you want to write your own pretty printers, refer to the gdb documentation. It is available online, for example, here: http://sourceware.org/gdb/current/onlinedocs/gdb/

Some pretty printers may occasionally interfere with the debugged program (especially if the program's state is already corrupted by earlier errors). Therefore, at times, it may be useful to temporarily disable pretty printers. To prevent pretty printers from being loaded for a session, clear the *GDB command file* setting in the launch configuration. To disable them for a currently active debug session, switch to the *gdb* page in the *Console* and enter the following gdb command:

```
disable pretty-printer global
```

Or, to only disable OMNEST-specific pretty printers (but leave the standard C++ library printers on):

```
disable pretty-printer global omnetpp;.*
```

6.6 Just-in-Time Debugging

The OMNEST runtime has the ability to launch an external debugger and attach it to the simulation process. You can configure a simulation to launch the debugger immediately on startup or when an error (runtime error or crash) occurs. This just-in-time debugging facility was primarily intended for use on Linux.

To enable just-in-time debugging, set the `debugger-attach-on-startup` or `debugger-attach-on-error` configuration option to `true`. You can do this by adding the appropriate line to `omnetpp.ini` or specifying `--debugger-attach-on-startup=true` in the *Additional arguments* field in the launch configuration dialog. You can also configure the debugger command line.

Note: On some systems (e.g., Ubuntu), just-in-time debugging requires extra setup beyond installing an external debugger. See the install-guide for more details.

6.7 Profiling a Simulation on Linux

On Linux systems, the OMNEST IDE supports executing your simulation using the *valgrind* program. Running your program with valgrind allows you to find memory-related issues and programming errors in your code. The simulation will run in an emulated environment (much slower than normal execution speeds), but valgrind will generate a detailed report when it finishes. The report is shown in a separate *Valgrind View* at the end of the simulation run. The OMNEST IDE contains support only for the `memcheck` tool. If you want to use other tools (`cachegrind`, `callgrind`, `massif`, etc.), you may try to install the full 'Linux Tools Project' from the Eclipse Marketplace.

To start profiling, right-click on your project in the *Project Explorer* tree and select *Profile As* → *OMNEST Simulation*. Valgrind must already be installed on your system.

Note: Simulation executes considerably slower than a normal run. Prepare for long run times or limit the simulation time in your .INI file. Statistical convergence is not required; just run long enough that all the code paths are executed in your model.

6.8 Controlling the Execution and Progress Reporting

After starting a simulation process or simulation batch, you can keep track of the started processes in the *Debug View*. To open the *Debug View* automatically during launch, check the "Show Debug View on Launch" option in the run configuration dialog, or select *Window* → *Show View* → *Other* → *Debug* → *Debug*. Select a process and click the terminate button to stop a specific simulation run, or use the context menu for more options to control the process execution.

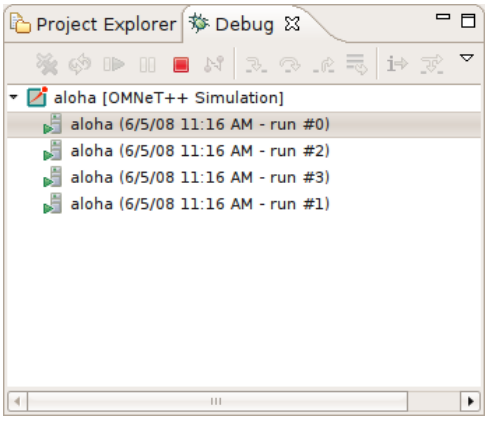


Fig. 6.5: Debug View

Tip: Place the Debug View in a different tab group than the console so you will be able to switch between the process outputs and see the process list at the same time.

Note: You can terminate all currently running processes by selecting the root of the launch. This will not cancel the entire batch, only the currently active processes. If you want to cancel the whole batch, open the *Progress View* and cancel the simulation batch there.

Clicking on a process in the *Debug View* switches to the output of that process in the *Console View*. The process may request user input via the console as well. Switch to the appropriate console and provide the requested parameters.

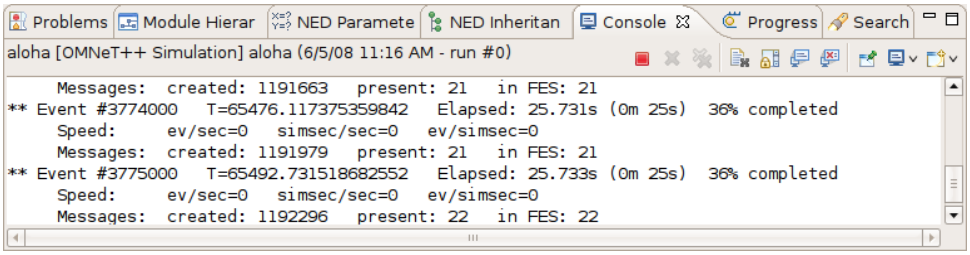


Fig. 6.6: Displaying the output of a simulation process in the Console View

---

Note: By default, the *Console View* automatically activates when a process writes to it. If you are running several parallel processes, this behavior might be inconvenient and prevent you from switching to the *Progress View*. You can turn off auto-activation by disabling the *Show Console When Standard Out/Error Changes* option in the *Console View* toolbar.

6.8.1 Progress Reporting

If you have executed the simulation in the command-line environment, you can monitor the progress of the simulation in the *Progress View*. See the status line for the overall progress indicator, and click on it to open the detailed progress view. You can terminate the entire batch by clicking on the cancel button in the *Progress View*.

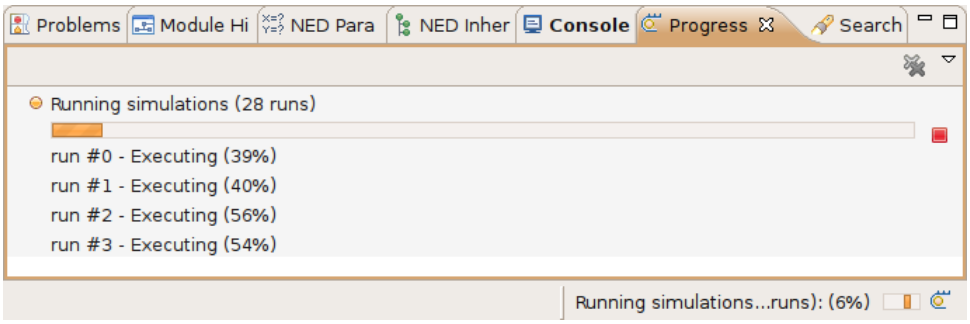


Fig. 6.7: Progress report on four parallel processes

Note: When the *Progress View* displays "Waiting for user input", the simulation is waiting for user input. Switch to the appropriate console and provide the requested input for the simulation.

Note: If you need more frequent progress updates, set the `cmdenv-status-frequency` option in your INI file to a lower value.

CHAPTER

SEVEN

THE QTENV GRAPHICAL RUNTIME ENVIRONMENT

7.1 Features

Qtenv is a graphical runtime interface for simulations. Qtenv supports interactive simulation execution, animation, inspection, tracing, and debugging. In addition to model development and verification, Qtenv is also useful for presentation and educational purposes, as it allows the user to get a detailed picture of the state and history of the simulation at any point of its execution.

When used together with a C++ source-level debugger, Qtenv can significantly speed up model development.

Its most important features are:

- network visualization
- message flow animation
- various run modes: event-by-event, normal, fast, express
- run until (a scheduled event, any event in a module, or given simulation time)
- simulation can be restarted
- a different configuration/run or network can be set up
- log of message flow
- display of textual module logs
- inspectors for viewing contents of objects and variables in the model
- eventlog recording for later analysis
- capturing a video of the main window
- snapshots (detailed report about the model: objects, variables, etc.)

7.2 Overview of the User Interface

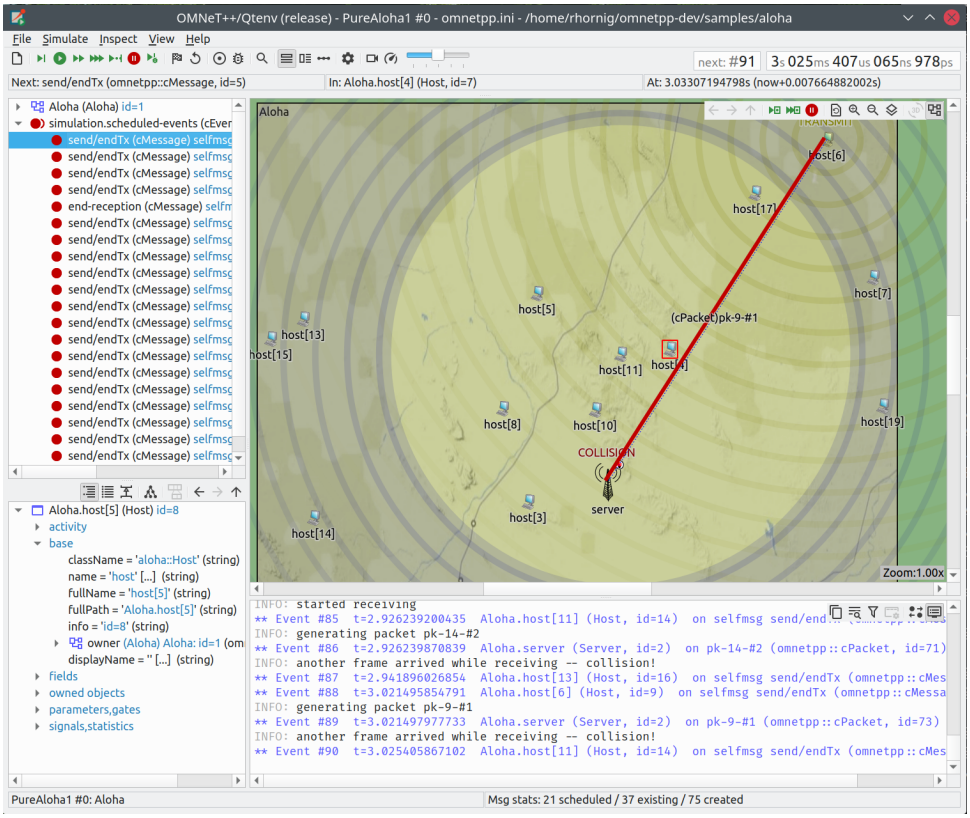


Fig. 7.1: The main window of Qtenv

Note: If you are experiencing graphics glitches, unreadable text, or the desktop color scheme you have set up is not suitable for Qtenv, you can disable the platform integration style plugins of Qt by setting the `QT_STYLE_OVERRIDE` environment variable to `fusion`. This will make the widgets appear in a platform-independent manner, as shown above.

The top of the window contains the following elements below the menu bar:

- *Toolbar*: The toolbar lets you access the most frequently used functions, such as stepping, running and stopping the simulation.
- *Animation speed*: The slider at the end of the toolbar lets you scale the speed of the built-in animations, as well as the playback speed of the custom animations added by the model.
- *Event Number and Simulation Time*: These two labels at the right end of the toolbar display the event number of the last executed or the next future event, and the current simulation time. The display format can be changed from the context menu.
- *Top status bar*: Three labels in a row that display either information about the next simulation event (in *Step* and *Normal* mode), or performance data like the number of events processed per second (in *Fast* and *Express* mode). This can be hidden to free up vertical space.
- *Timeline*: Displays the contents of the Future Events Set (FES) on a logarithmic time scale. The timeline can be turned off to free up vertical space.
- *Bottom status bar*: Displays the current configuration, the run number, and the name

of the root module (network) on the left, and a few statistics about the message objects in the model on the right.

The central area of the main window is divided into the following regions:

- *Object Navigator*: Displays the hierarchy of objects in the current simulation and in the FES.
- *Object Inspector*: Displays the contents and properties of the selected object.
- *Network Display*: Displays the network or any module graphically. This is also where animation takes place.
- *Log Viewer*: Displays the log of packets or messages sent between modules, or log messages output by modules during simulation.

Additionally, you can open inspector windows that float on top of the main window.

7.3 Using Qtenv

7.3.1 Starting Qtenv

When you launch a simulation from the IDE, it will be started with Qtenv by default. When it does not, you can explicitly select Qtenv in the *Run* or *Debug* dialog.

Qtenv is also the default when you start the simulation from the command line. When necessary, you can force Qtenv by adding the `-u Qtenv` switch to the command line.

The complete list of command-line options, related environment variables, and configuration options can be found at the end of this chapter.

7.3.2 Setting Up and Running the Simulation

On startup, Qtenv reads the ini file(s) specified on the command line (or `omnetpp.ini` if none is specified), and automatically sets up the simulation described in them. If they contain several simulation configurations, Qtenv will ask you which one you want to set up.

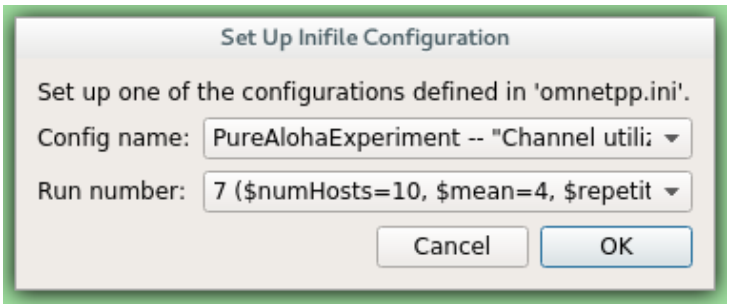


Fig. 7.2: Setting Up a New Simulation

Once a simulation has been set up (modules have been created and initialized), you can run it in various modes and examine its state. You can restart the simulation at any time, or set up another simulation. If you choose to quit Qtenv before the simulation finishes (or try to restart the simulation), Qtenv will ask you whether to finalize the simulation, which usually translates to saving summary statistics.

Functions related to setting up a simulation are in the *File* and *Simulate* menus, and the most important ones are accessible via toolbar icons and keyboard shortcuts.

Some of these functions are:

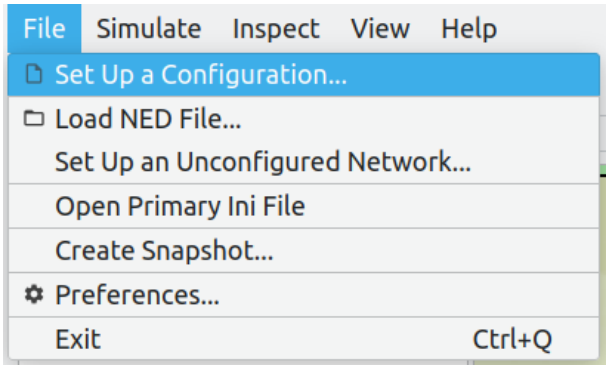


Fig. 7.3: The File menu

Set up a Configuration

This function lets you choose a configuration and run number from the ini file.

Open Primary Ini File

Opens the first ini file in a text window for viewing.

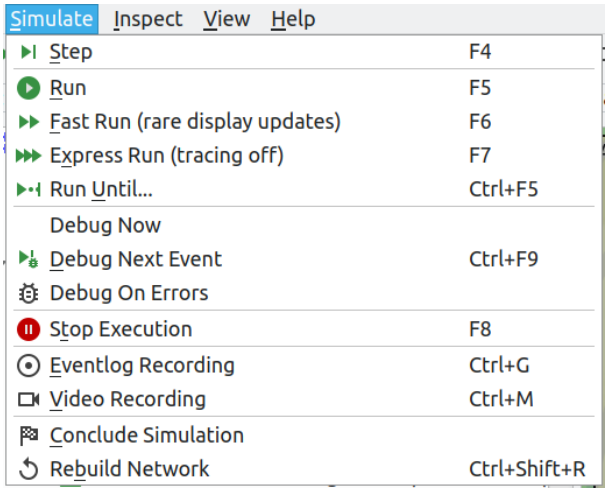


Fig. 7.4: The Simulate menu

Step

Step lets you execute one simulation event, which is at the front of the FES. The next event is always shown on the status bar. The module where the next event will be delivered is highlighted with a red rectangle on the graphical display.

Run (or Normal Run)

In *Run* mode, the simulation runs with all tracing aids on. Message animation is active, and the simulation time is interpolated if the model requested a non-zero animation speed. Inspector windows are constantly updated. Output messages are displayed in the main window and module output windows. You can stop the simulation with the *Stop* button on the toolbar. You can fully interact with the user interface while the simulation is running, such as opening inspectors.

Note: If you find this mode too slow or distracting, you may switch off animation features in the *Preferences* dialog.

Fast Run

In *Fast* mode, message animation is turned off. The inspectors are updated much less often. Fast mode is several times faster than the Run mode; the speed can increase by up to 10 times (or up to the configured event count).

Express Run

In *Express* mode, the simulation runs at about the same speed as with Cmdenv, with all tracing disabled. Module log is not recorded. The simulation can only be interacted with once in a while, so the run-time overhead of the user interface is minimal. UI updates can even be disabled completely, in which case you have to explicitly click the *Update now* button to refresh the inspectors.

Run Until

You can run the simulation until a specified simulation time, event number, or until a specific message has been delivered or canceled. This is a valuable tool during debugging sessions. It is also possible to right-click on an event in the simulation timeline and choose the *Run until this event* menu item.

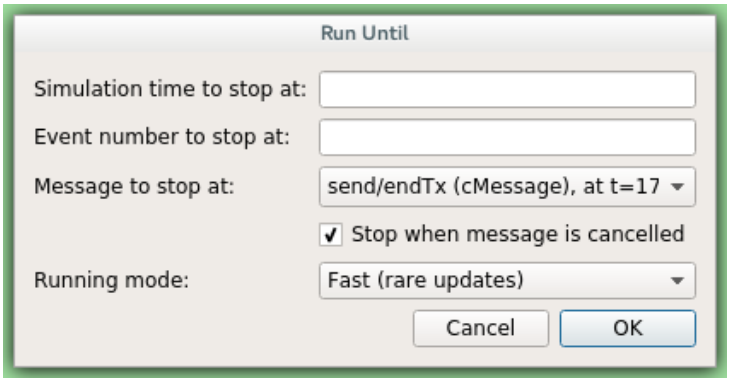


Fig. 7.5: The Run Until dialog

Run Until Next Event

It is also possible to run until an event occurs in a specified module. Browse for the module and choose *Run until next event in this module.* Simulation will stop once an event occurs in the selected module.

Debug Next Event

This function is useful when you are running the simulation under a C++ source-level debugger. *Debug Next Event* will perform one simulation event just like *Step*, but executes a software debugger breakpoint (`int3` or `SIGTRAP`) just before entering the module's event handling code (`handleMessage()` or `activity()`). This will cause the debugger to stop the program there, allowing you to examine state variables, single-step, etc. When you resume execution, Qtenv will regain control and become responsive again.

Debug On Errors

This menu item allows you to change the value of the `debug-on-errors` configuration variable on the fly. This is useful if you forgot to set this option before starting the simulation, but would like to debug a runtime error. The state of this menu item is reset to the value of `debug-on-errors` every time Qtenv is started.

Recording an Event Log

The OMNEST simulation kernel allows you to record event-related information into a file, which can later be used to analyze the simulation run using the *Sequence Chart* tool in the IDE. Eventlog recording can be turned on with the `record-eventlog=true` ini file option, but also interactively, via the respective item in the *Simulate* menu, or using a toolbar button.

Note that starting Qtenv with `record-eventlog=true` and turning on recording later does not result in exactly the same eventlog file. In the former case, all steps of setting up the network, such as module creations, are recorded as they happen; while for the latter, Qtenv has to "fake" a chain of steps that would result in the current state of the simulation.

Capturing a Video

When active, this feature will save the contents of the main window into a subfolder named `frames` in the working directory with a regular frequency (in animation time). Each frame is a PNG image, with a sequence number in its file name. Currently, the user has to convert (encode) these images into a video file after the fact by using an external tool (such as `ffmpeg`, `avconv`, or `vlc`). When the recording is started, an info dialog pops up, showing further details on the output, and an example command for encoding in high quality using `ffmpeg`. The resulting video is also affected by the speed slider on the toolbar.

Note: This built-in recording feature is able to produce a smooth video, in contrast to external screen-capture utilities. This is possible because it has access to more information and has more control over the process than external tools.

Conclude Simulation

This function finalizes the simulation by invoking the user-supplied `finish()` member functions on all module and channel objects in the simulation. The customary implementation of `finish()` is to record summary statistics. The simulation cannot be continued afterwards.

Rebuild Network

Rebuilds the simulation by deleting the current network and setting it up again. Improperly written simulations often crash when *Rebuild Network* is invoked, usually due to incorrectly written destructors in module classes.

7.3.3 Inspecting Simulation Objects

Inspectors

The *Network Display*, the *Log Viewer*, and the *Object Inspector* in the main window share some common properties: they display various aspects (graphical view / log messages / fields or contents) of a given object. Such UI parts are called *inspectors* in Qtenv.

The three inspectors mentioned above are built into the main window, but you can open additional ones at any time. The new inspectors will open in floating windows above the main window, and you can have any number of them open.

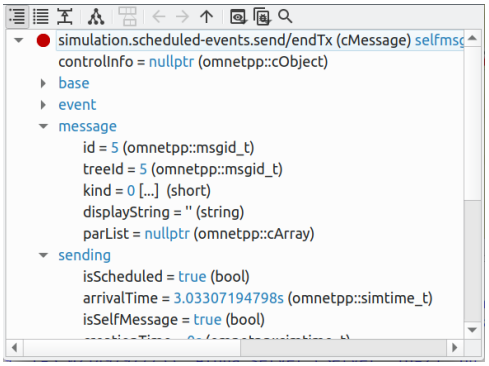


Fig. 7.6: A floating inspector window

Inspectors come in many flavors. They can be graphical like the network view, textual like the log viewer, tree-based like the object inspector, or something entirely different.

Note: Some window managers might disable/hide the close button of floating inspectors. If this happens, you can still close them with a keyboard shortcut (most commonly Alt + F4), or by right-clicking on the title bar, and choosing the Close option in the appearing menu.

Opening Inspectors

Inspectors can be opened in various ways: by double-clicking an item in the *Object Navigator* or in other inspectors; by choosing one of the *Open* menu items from the context menu of an object displayed on the UI; via the *Find/Inspect Objects* dialog (see later); or even by directly entering the C++ pointer of an object as a hex value. Inspector-related menu items are in the *Inspect* menu.

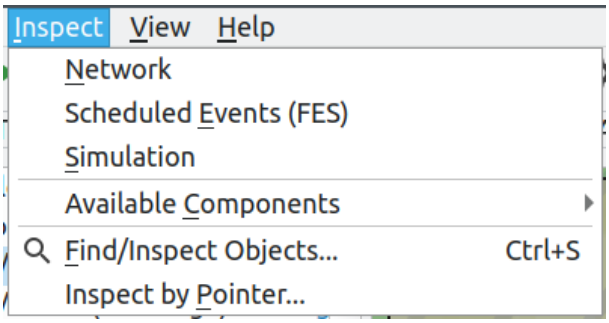


Fig. 7.7: The Inspect menu

History

Inspectors always show some aspect of one simulation object, but they can change objects. For example, in the *Network View*, when you double-click a submodule that is itself a compound module, the view will switch to showing the internals of that module; or, the *Object Inspector* will always show information about the object last clicked in the UI. Inspectors maintain a navigable history: the *Back/Forward* functions go to the object inspected before/after the currently displayed object. Objects that are deleted during simulation also disappear from the history.

Restoring Inspectors

When you exit and then restart a simulation program, Qtenv tries to restore the open inspector windows. However, as object identity is not preserved across different runs of the same program, Qtenv uses the object full path, class name, and object ID (where exists) to find and identify the object to be inspected.

Preferences such as zoom level or open/closed state of a tree node are usually maintained per object type (i.e. tied to the C++ class of the inspected object).

Extending Qtenv

It is possible for the user to contribute new inspector types without modifying Qtenv code. For this, the inspector C++ code needs to include Qtenv header files and link with the Qtenv library. One caveat is that the Qtenv headers are not public API and thus subject to change in a new version of OMNEST.

7.4 Using Qtenv with a Debugger

You can use Qtenv in combination with a C++ debugger, which is mainly useful when developing new models. When doing so, there are a few things you need to know.

Qtenv is a library that runs as part of the simulation program. This has several implications, the most apparent being that when the simulation crashes (due to a bug in the model's C++ code), it will bring down the whole OS process, including the Qtenv GUI.

The second consequence is that suspending the simulation program in a debugger will also freeze the GUI until it is resumed. Also, Qtenv is single-threaded and runs in the same thread as the simulation program, so even if you only suspend the simulation's thread in the debugger, the UI will freeze.

The Qtenv UI deals with `cObject`s (the C++ methods that the GUI relies on are defined on `cObject`). All other data such as primitive variables, non-`cObject` classes and structs, STL containers, etc., are hidden from Qtenv. You may wrap objects into `cObject`s to make them visible for Qtenv; that's what the `WATCH` macros do as well.

The following sections go into detail about various parts and functions of the Qtenv UI.

7.5 Parts of the Qtenv UI

7.5.1 The Status Bars

The status bars show the simulation's progress. There is one row at the top of the main window, and one at the bottom. The top one can be hidden using the *View* → *Status Details* menu item.

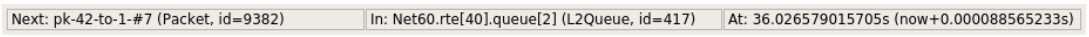


Fig. 7.8: The top status bar

When the simulation is paused or runs with animation, the top row displays the next expected simulation event. Note the word *expected*. Certain schedulers may insert new events before the displayed event at the last moment. Some schedulers that tend to do that are those that accept input from outside sources: real-time scheduler, hybrid or hardware-in-the-loop schedulers, parallel simulation schedulers, etc. The top row contains the following:

1. Name, C++ class, and ID of the next message (event) object
2. The module where the next event will occur (i.e., the module where the message will be delivered)
3. The simulation time of the next (expected) simulation event
4. Time of the next event, and delta from the current simulation time

When the simulation is running in *Fast* or *Express* mode, displaying the next event becomes useless. The contents of the top row are replaced by the following performance gauges:

Fig. 7.9: The top status bar during Fast or Express run

1. Simulation speed: number of events processed per real second
2. Relative speed of the simulation (compared to real-time)
3. Event density: the number of events per simulated second

The bottom row contains the following items:

Fig. 7.10: The bottom status bar

1. Ini config name, run number, and the name of the network
2. Message statistics: the number of messages currently scheduled (i.e., in the FES); the number of message objects that currently exist in the simulation; and the number of message objects that have been created so far, including the already deleted ones. Out of the three, the middle one is probably the most useful. If it is steadily growing without apparent reason, the simulation model is probably missing some `delete msg` statements and needs debugging.

7.5.2 The Timeline

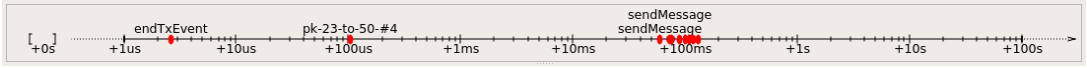


Fig. 7.11: The timeline

The timeline displays the contents of the Future Events Set on a logarithmic time scale. Each dot represents a message (event). Messages to be delivered in the current simulation time are grouped into a separate section on the left between brackets.

Clicking an event will focus it in the *Object Inspector*, and double-clicking it will open a floating inspector window. Right-clicking it will bring up a context menu with further actions.

The timeline is often crowded, limiting its usefulness. To overcome this, you can hide uninteresting events from the timeline. Right-click the event and choose *Exclude Messages Like 'x' From Animation* from the context menu. This will hide events with a similar name and the same C++ class name from the timeline, and also skip the animation when such messages are sent from one module to another. You can view and edit the list of excluded messages on the *Filtering* page of the *Preferences* dialog. (Tip: the timeline context menu provides a shortcut to that dialog).

The whole timeline can be hidden (and revealed again) using the *View* → *Timeline* menu item, by pressing a button on the toolbar, or simply by dragging the handle of the separator under it all the way up.

7.5.3 The Object Navigator

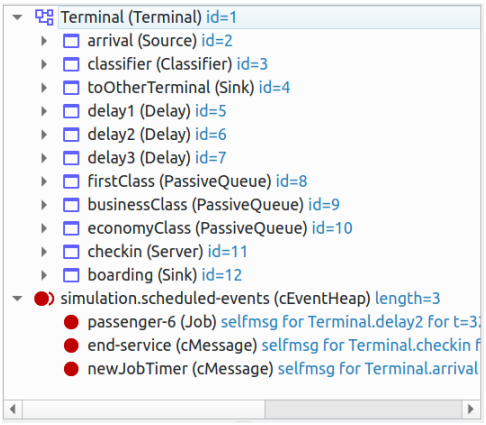


Fig. 7.12: The object tree

The *Object Navigator* displays the hierarchy of objects in the current simulation and in the FES in a tree form.

Clicking an object will focus it in the *Object Inspector*, and double-clicking it will open a floating inspector window. Right-clicking it will bring up a context menu with further actions.

7.5.4 The Object Inspector

The *Object Inspector* is located below the *Object Navigator* and lets you examine the contents of objects in detail. The *Object Inspector* always focuses on the object last clicked or selected on the Qtenv UI. It can also be navigated directly using the *Back*, *Forward*, and *Go to Parent* buttons, and by double-clicking objects shown inside the inspector's area.

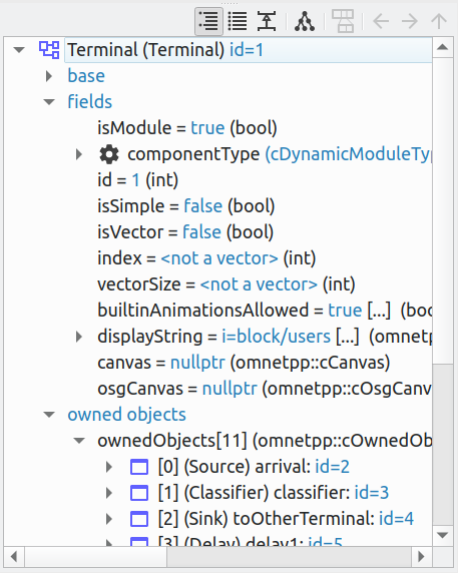


Fig. 7.13: The object inspector in Grouped mode

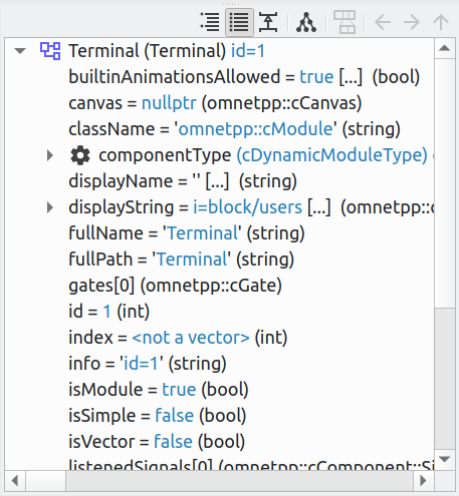


Fig. 7.14: The object inspector in Flat mode

The inspector has four display modes: *Grouped*, *Flat*, *Children*, and *Inheritance*. You can switch between these modes using the buttons on the inspector's toolbar.

In *Grouped*, *Flat*, and *Inheritance* modes, the tree shows the fields (or data members) of the object. It uses meta-information generated by the message compiler to obtain the list of fields and their values. (This is true even for the built-in classes – the simulation kernel contains their description of msg format.)

The only difference between these three modes is the way the fields are arranged. In *Grouped* mode, they are organized in categories; in *Flat* mode, they form a simple alphabetical list; and in *Inheritance* mode, they are organized based on which superclass they are inherited from.

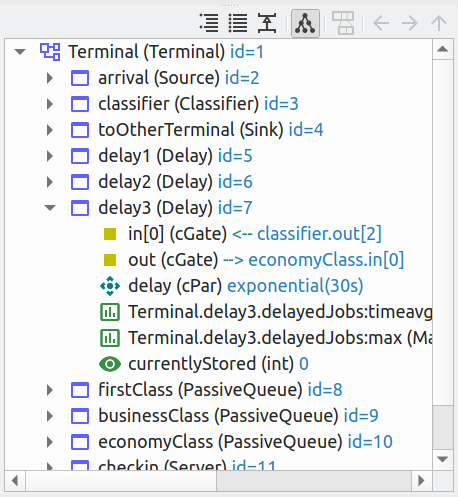


Fig. 7.15: The object inspector in Children mode

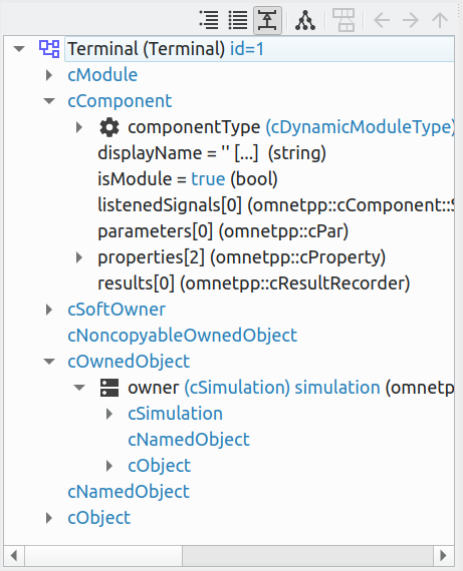


Fig. 7.16: The object inspector in Inheritance mode

In *Children* mode, the tree shows the child objects of the currently inspected object. The child list is obtained via the `forEachChild()` method of the object. This is very similar to how the *Object Navigator* works, but this can have an arbitrary root.

7.5.5 The Network Display

The network view provides a graphical view of the network and modules in general. The graphical representation is based on display strings (`@display` properties in the NED file). You can go into any compound module by double-clicking its icon.

Message sending, method calls, and certain other events are animated in the graphical view. You can customize animation in the *Animation* page of the *Preferences* dialog.

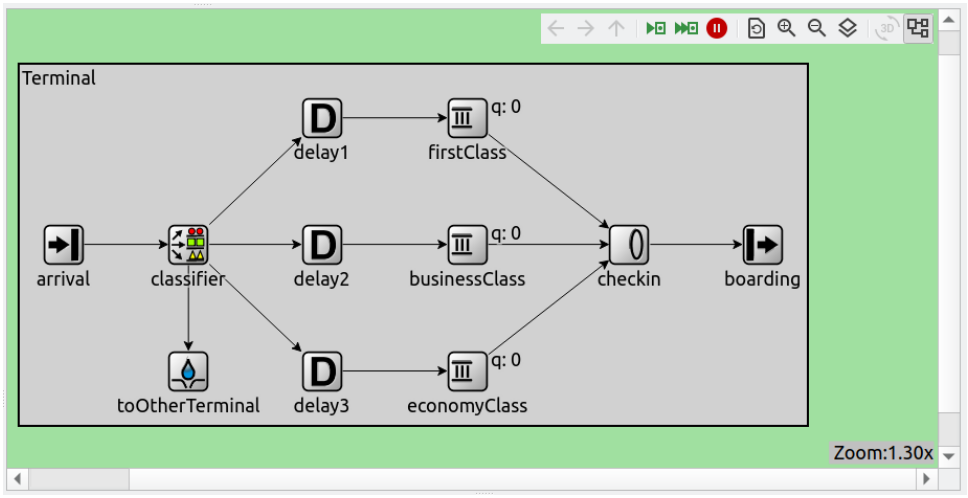


Fig. 7.17: The network display

The built-in `cCanvas` of the inspected object is also rendered in this view together with the module contents to allow overlaying custom annotations and animations. This canvas contains the figures declared by the `@figure` properties in the NED source of the module.

By choosing the *Show/Hide Canvas Layers* item in the context menu of the inspected module, you can filter the displayed figures based on the tags set on them.

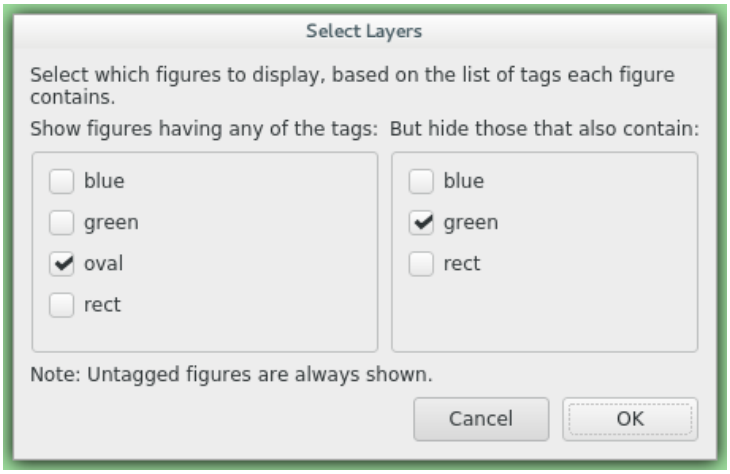


Fig. 7.18: Figure filtering dialog

Since any figure can have any number of tags, a two-step filtering mechanism is applied to provide sufficient control. The left side is a whitelist, while the right side is a blacklist. The example above would only let all the figures with the “oval” tag appear, except those that also have the “green” tag on them.

If the inspected module has a built-in `cOsgCanvas` (and Qtenv is built with OSG support enabled), this inspector can also be switched into a 3D display mode with the globe icon on its toolbar. In this case, the 2D network and canvas display is replaced by the scene encapsulated by the `cOsgCanvas`.

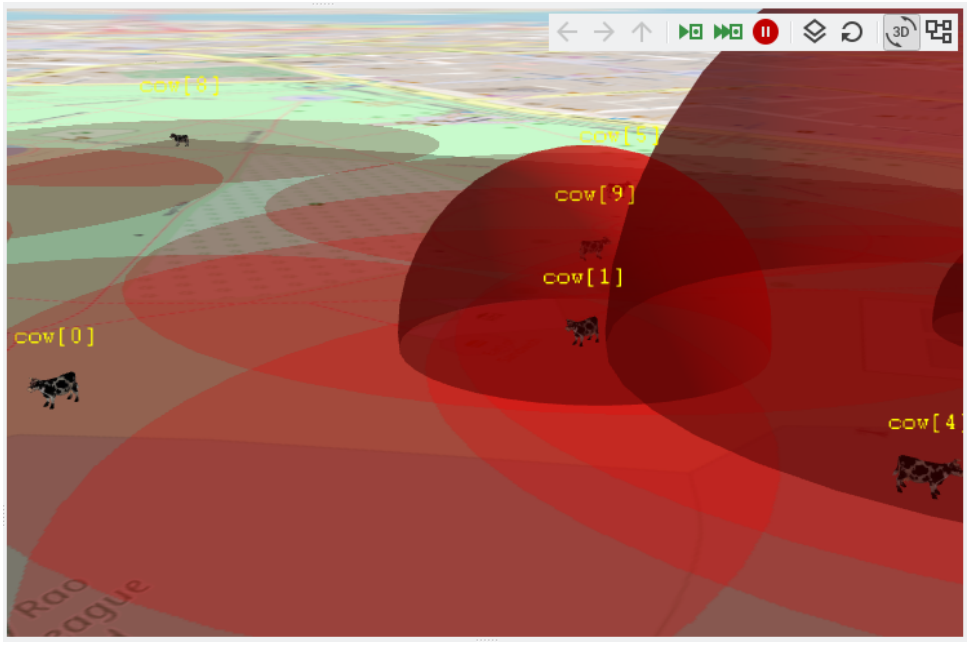


Fig. 7.19: The network display in 3D mode

The context menu of submodules provides further actions (see below).

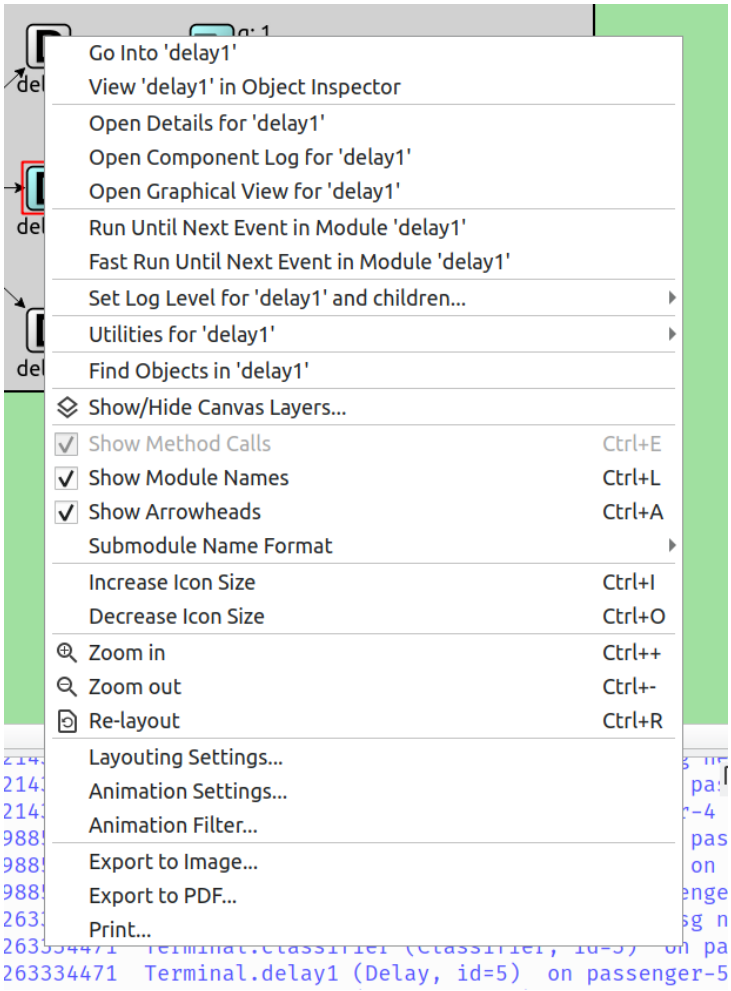


Fig. 7.20: Submodule context menu

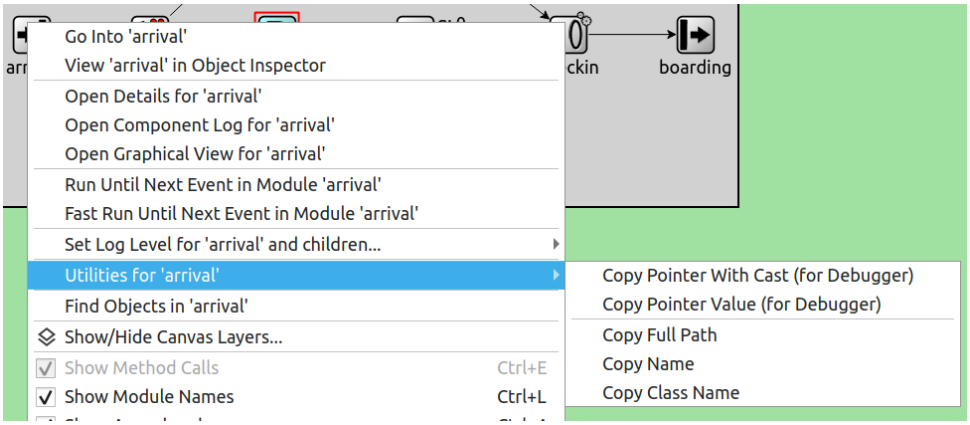


Fig. 7.21: The Utilities submenu

Zooming and Panning

There are several ways to zoom the canvas, both using the mouse and the keyboard:

- To zoom in around a point, double-click the canvas; use Shift + double-click to zoom out or scroll while holding down Ctrl.

  You can also zoom around the center of the viewport with the looking glass buttons on the canvas toolbar.

- For marquee zoom, drag out a rectangle with the left mouse button while holding down Ctrl; you can cancel the operation with the right mouse button.

- Panning: moving the mouse while holding down the left mouse button will move the canvas; this is often a more comfortable way to navigate the canvas than using the scroll bars. You can of course scroll in any direction with simply the mouse wheel or the similar functionality of many touchpads.

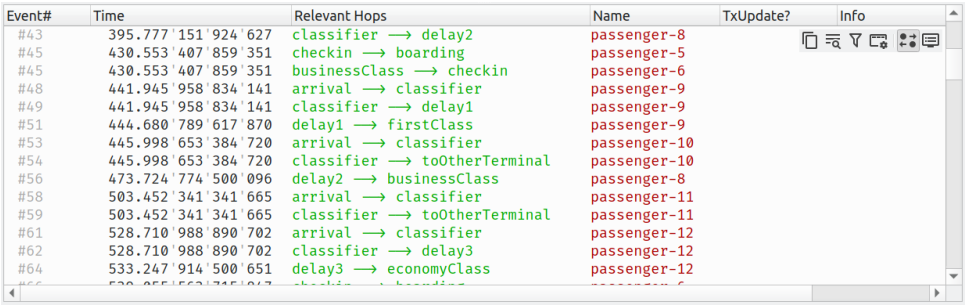
7.5.6 The Log Viewer

When you run the simulation, Qtenv will remember the output from logging statements (`EV << "Hello World\n";`) and the messages sent between modules, and can present it to you in a meaningful manner. Only the output from the last N events is preserved (N being configurable in the *Preferences* dialog), and only in Step, Run, and Fast Run modes. (Express mode can be so fast because such overhead is turned off while it's active.)

The *Log Viewer* shows log related to one compound module and its subtree. It has two modes: *Messages* and *Log* mode, with *Messages* being the default. You can switch between the two modes using tool icons on the inspector's local toolbar.

In *Messages* mode, the window displays messages sent between the (immediate) submodules of the inspected compound module, and messages sent out of or into the compound module. The embedded *Log Viewer* shows content related to the module inspected in the *Network Display* above it at any time. You can view details about any message in the *Object Inspector* by clicking on it, and access additional functions in its context menu.

Note: In *Messages* mode, the *Info* column can be customized by writing and registering a custom `cMessagePrinter` class. This string is split at the tab characters (`'\t'`) into parts that are aligned in additional columns.



Event#	Time	Relevant Hops	Name	TxUpdate?	Info
#43	395.777'151'924'627	classifier ⟶ delay2	passenger-8		
#45	430.553'407'859'351	checkin ⟶ boarding	passenger-5		
#45	430.553'407'859'351	businessClass ⟶ checkin	passenger-6		
#48	441.945'958'834'141	arrival ⟶ classifier	passenger-9		
#49	441.945'958'834'141	classifier ⟶ delay1	passenger-9		
#51	444.680'789'617'870	delay1 ⟶ firstClass	passenger-9		
#53	445.998'653'384'720	arrival ⟶ classifier	passenger-10		
#54	445.998'653'384'720	classifier ⟶ toOtherTerminal	passenger-10		
#56	473.724'774'500'096	delay2 ⟶ businessClass	passenger-8		
#58	503.452'341'341'665	arrival ⟶ classifier	passenger-11		
#59	503.452'341'341'665	classifier ⟶ toOtherTerminal	passenger-11		
#61	528.710'988'890'702	arrival ⟶ classifier	passenger-12		
#62	528.710'988'890'702	classifier ⟶ delay3	passenger-12		
#64	533.247'914'500'651	delay3 ⟶ economyClass	passenger-12		

Fig. 7.22: The log viewer showing message traffic

In *Log* mode, the window displays log lines that belong to submodules under the inspected compound module (i.e., the whole module subtree).

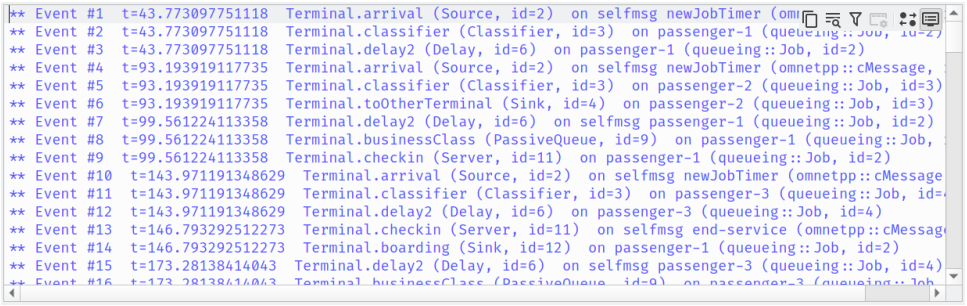


Fig. 7.23: The log viewer showing module log

You can filter the content of the window to only include messages from specific modules. Open the log window's context menu and select *Filter Window Contents*.

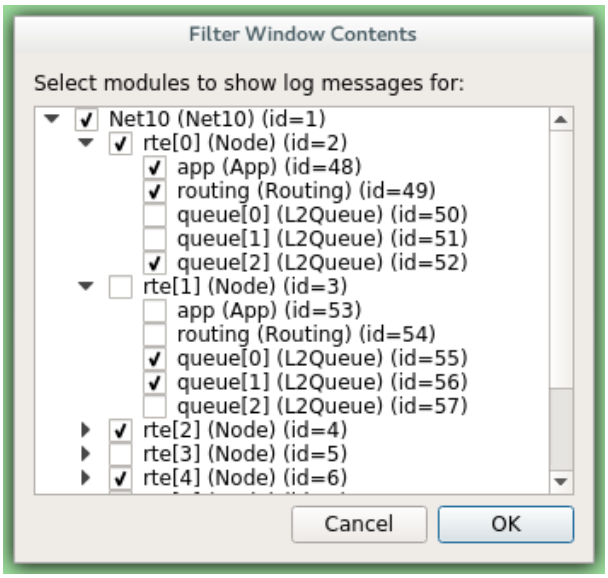


Fig. 7.24: The log filter dialog

General logging behavior, such as the prefix format, can be controlled in the *Preferences* dialog. The log level of each module (and its descendants) can be set in its context menu.

It is also possible to open separate log windows for individual modules. A log window for a compound module displays the log from all of its submodule tree. To open a log window, find the module in the module tree or the network display, right-click it, and choose *Open Component Log* from the context menu.

7.6 Inspecting Objects

7.6.1 Object Inspectors

In addition to the inspectors embedded in the main window, Qtenv lets you open floating inspector windows for individual objects. The screenshot below shows Qtenv with several inspectors open.

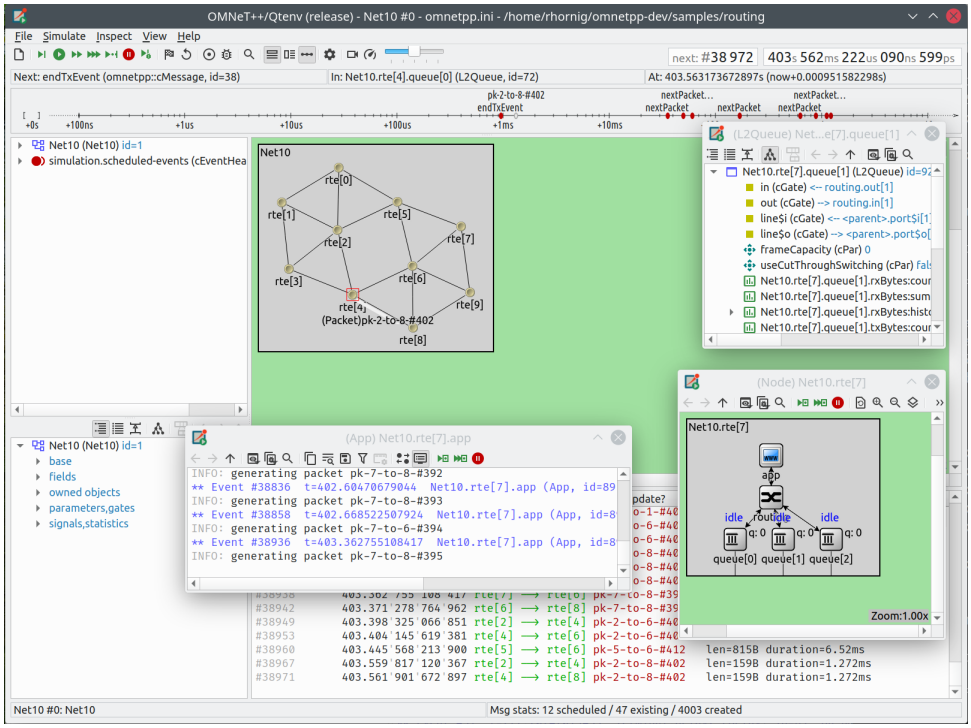


Fig. 7.25: Qtenv with several floating inspectors open

7.6.2 Browsing the Registered Components

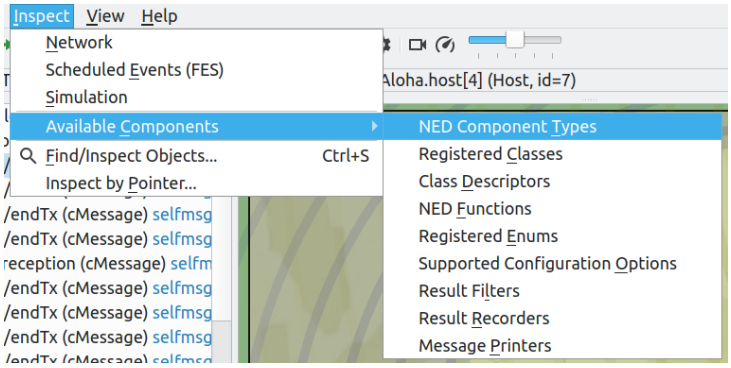


Fig. 7.26: The Inspect menu

Registered components (NED Types, classes, functions, enums) can be displayed with the *Inspect → Available components* menu item. If an error message reports missing types or classes, you can check here whether the missing item is in fact available, i.e., registered correctly.

7.6.3 Querying Objects

The *Find/Inspect Objects* dialog allows you to search the simulation for objects that meet certain criteria. The criteria can be the object name, class name, the value of a field of the object, or a combination of those. The results are presented in a table that can be sorted by columns, and items in the table can be double-clicked to inspect them.

Some possible use cases:

- Identifying bottlenecks in the network by looking at the list of all queues and ordering them by length (i.e. having the result table sorted by the *Info* column)
- Finding nodes with the highest packet drop count. If the drop counts are watched variables (see the `WATCH()` macro), you can get a list of them.
- Finding modules that leak messages. If the live message count on the status bar keeps increasing, you can search for all message objects and see where the leaked messages are hiding.
- Easy access to some data structures or objects, such as routing tables. You can search by name or class name and use the result list as a collection of hotlinks, saving you from manually navigating the simulation's object tree.

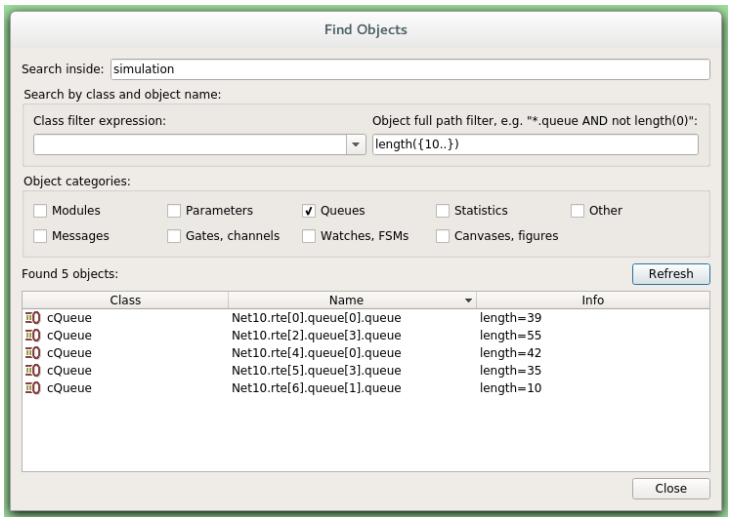


Fig. 7.27: Using the Find/Inspect Objects dialog to find long queues

The dialog allows you to specify the search root, as well as the name and class name of the objects to find. The latter two accept wildcard patterns.

The checkboxes in the dialog can be used to select the object categories that interest you. If you select a category, all objects of that type (and any types derived from it) will be included in the search. Alternatively, if you specify an object class as a class filter expression, the search dialog will try to match the object's class name with the given string. Objects of derived types will not be included in the search.

You can also provide a generic filter expression, which matches the object's full path by default. Wildcards (`"?"`, `"*"`) are allowed. `"{a-exz}"` matches any character in the range `"a"` through `"e"` and the characters `"x"` and `"z"`. You can match numbers using patterns like `"*.job{128..191}"` to match objects named `"job128"`, `"job129"`, and `"job191"`. You can also use patterns like `"job{128..}"` and `"job{..191}"`. Patterns can be combined using `AND`, `OR`, `NOT`, and parentheses. Lowercase versions of these keywords (`and`, `or`, `not`) are also accepted. You can match other object fields, such as queue length, message kind, etc., using the syntax `fieldname =~ pattern`. If the pattern contains special characters or spaces, you need to enclose it in quotes. (HINT: In most cases, you will want to start the pattern with `"*."` to match objects anywhere in the network!).

Examples:

- `*.destAddr`: Matches all objects with the name `"destAddr"` (likely module parameters).
- `*.node[8..10].*`: Matches anything inside module `node[8]`, `node[9]`, and `node[10]`.
- `className =~ omnetpp::cQueue AND NOT length =~ 0`: Matches non-empty queue objects.
- `className =~ omnetpp::cQueue AND length =~ {10..}`: Matches queue objects with length greater than or equal to 10.
- `kind =~ 3 OR kind =~ {7..9}`: Matches messages with message kind equal to 3, 7, 8, or 9 (only messages have a `"kind"` attribute).
- `className =~ IP* AND *.data-*`: Matches objects whose class name starts with `"IP"` and name starts with `"data-"`.
- `NOT className =~ omnetpp::cMessage AND byteLength =~ {1500..}`: Matches messages whose class is not cMessage and byteLength is at least 1500 (only messages have a `"byteLength"` attribute).
- `"TCP packet" OR "*.packet(15)"`: Quotation marks are needed when the pattern is a reserved word or contains whitespace or special characters.

Note: Qtenv uses the `cObject::forEachChild` method to recursively collect all objects from a tree. If you have your own objects derived from `cObject`, you should redefine the `cObject::forEachChild` method to ensure correct object search functionality.

Note: The class names must be fully qualified, meaning they should contain the namespace(s) they are in, regardless of the related setting in the *Preferences dialog*.

Note: If you are debugging the simulation with a source level debugger, you can also use the *Inspect by pointer* menu item. Let the debugger display the address of the object you want to inspect, and paste it into the dialog. Please note that entering an invalid pointer will crash the simulation.

7.7 The Preferences Dialog

Select *File* → *Preferences* from the menu to display the runtime environment's configuration dialog. The dialog allows you to adjust various display, network layouting, and animation options.

7.7.1 General

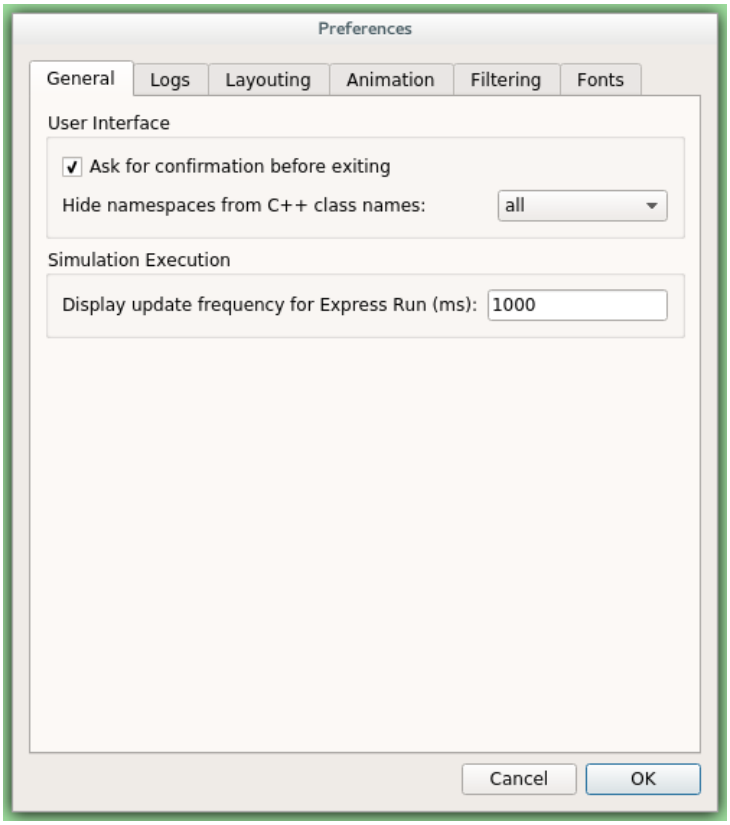


Fig. 7.28: General settings

The *General* tab can be used to set the default user interface behavior. You can choose whether namespaces should be stripped off the displayed class names, and how often the user interface should be updated while the simulation runs in *Express* mode.

7.7.2 Logs

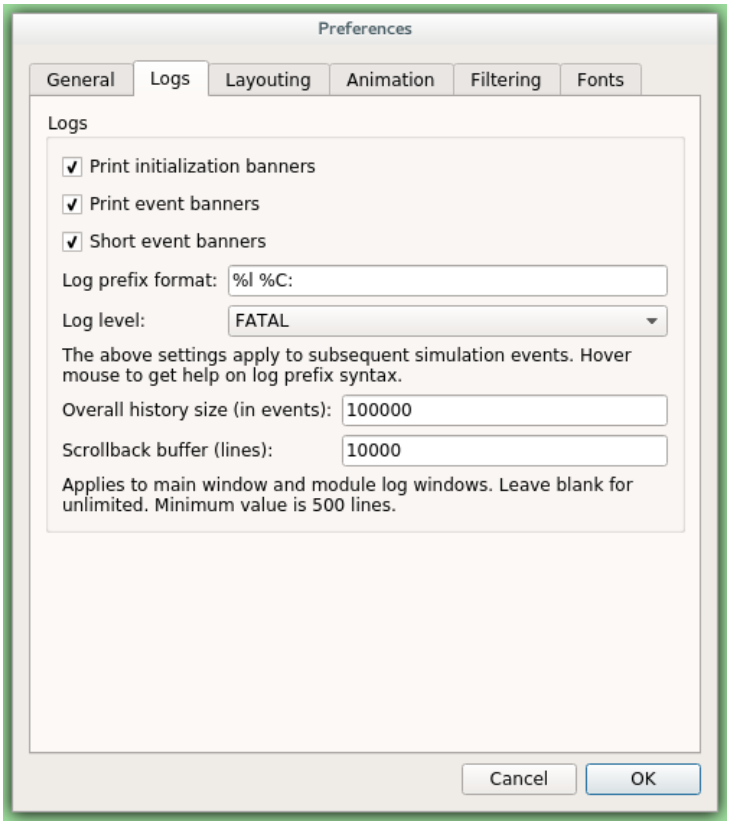


Fig. 7.29: Logging settings

The *Logs* tab can be used to set the default logging behavior, such as the log level of modules that do not override it, the prefix format for event banners, and the size limit of the log buffer.

7.7.3 Configuring the Layouting Algorithm

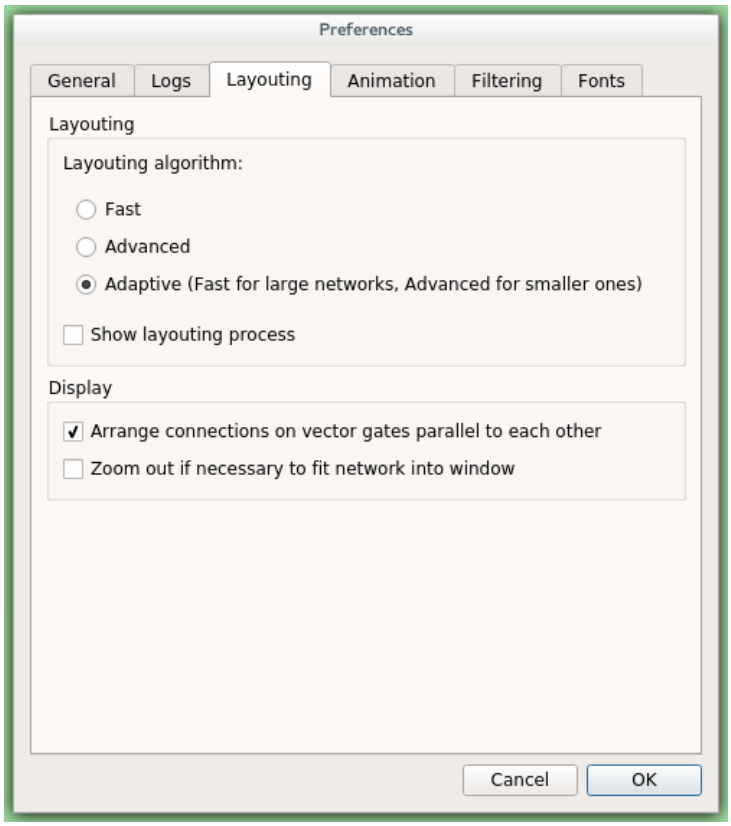


Fig. 7.30: Layouting settings

Qtenv provides automatic layouting for submodules that do not have their locations specified in the NED files. The layouting algorithm can be fine-tuned on the *Layouting* page of this dialog.

7.7.4 Configuring Animation

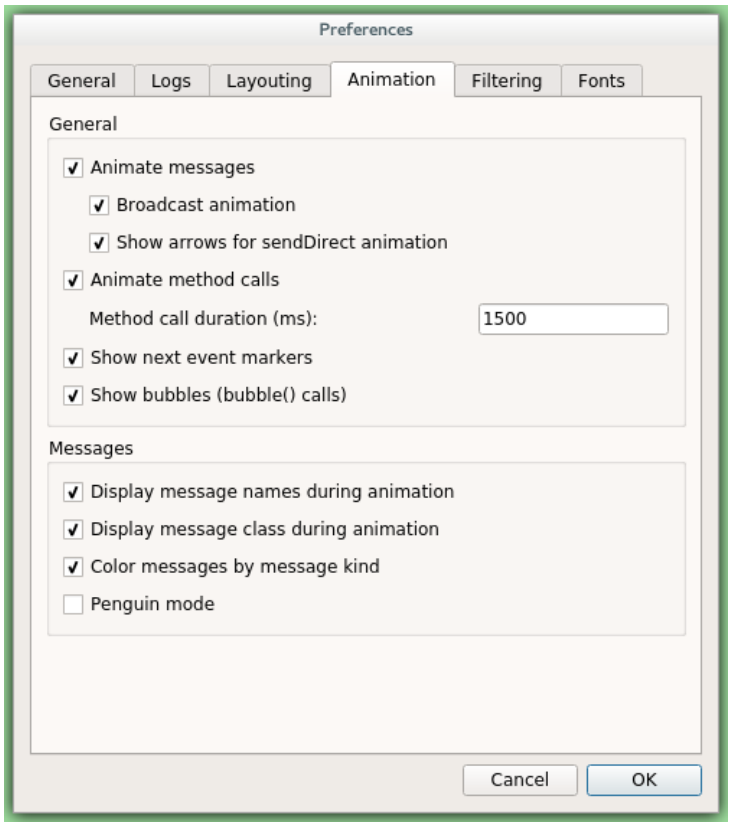


Fig. 7.31: Animation settings

Qtenv provides automatic animation when you run the simulation. You can fine-tune the animation settings using the *Animation* page of the settings dialog. If you do not need all of the visual feedback that Qtenv provides, you can selectively turn off some of the features:

- Animate messages: Turns on/off the visualization of messages passing between modules.
- Broadcast animation: Handles message broadcasts in a special way (zero-time messages sent within the same event will be animated concurrently).
- Show next event marker: Highlights the module that will receive the next event.
- Show a dotted arrow when a `sendDirect()` method call is executed.
- Show a flashing arrow when a method call occurs from one module to another. The call is only animated if the called method contains the `Enter_Method()` macro.
- The display of message names and classes can also be turned off.

7.7.5 Timeline and Animation Filtering

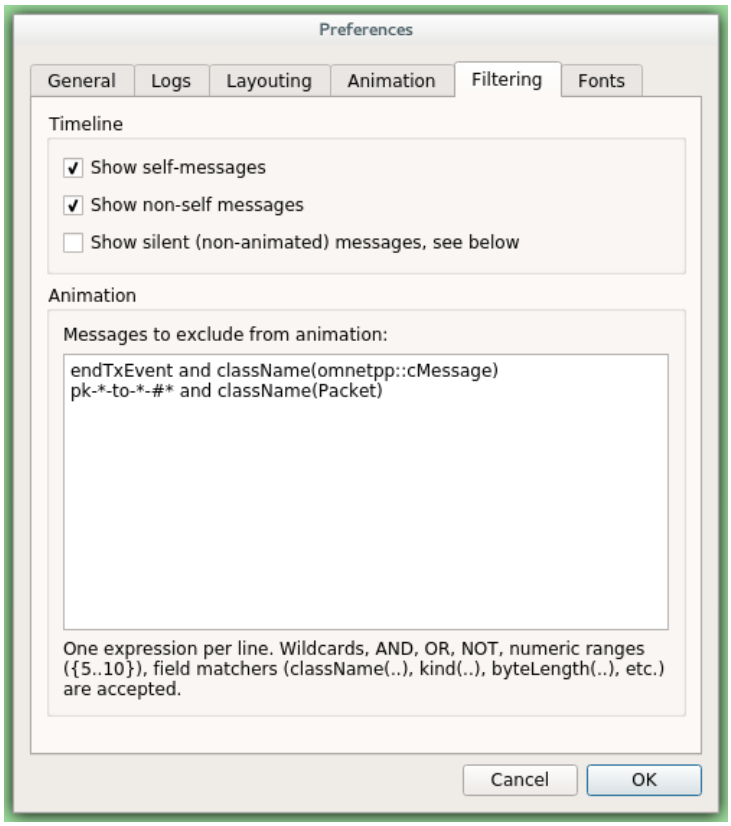


Fig. 7.32: Filtering

The *Filtering* page of the dialog serves two purposes. First, it allows you to filter the contents of the *Timeline*. You can hide all self-messages (timers) or all non-self messages. Additionally, you can further reduce the number of messages shown on the timeline by hiding the non-animated messages, as explained below.

Second, you can suppress the animation of certain messages. For example, when you are focused on routing protocol messages, you can suppress the animation of data traffic.

The text box allows you to specify multiple filters, with each filter on a separate line. You can filter messages by name, class name, or any other property that appears in the *Fields* page of the *Object Inspector* when focusing it on the given message object.

Note: When you select *Exclude Messages Like 'x' From Animation* from the context menu of a message object in the UI, it will add a new filter on this dialog page.

For object names, you can use wildcards (`"?"`, `"*"`). `"{a-exz}"` matches any character in the range `"a"` through `"e"` and the characters `"x"` and `"z"`. You can match numbers using patterns like `"job{128..191}"` to match `"job128"`, `"job129"`, and `"job191"`. You can also use patterns like `"job{128..}"` and `"job{..191}"`. Patterns can be combined using `AND`, `OR`, `NOT`, and parentheses. Lowercase versions of these keywords (`and`, `or`, `not`) are also accepted. You can match against other object fields, such as message length, message kind, etc., using the syntax `fieldname =~ pattern`. If the pattern contains special characters or spaces, you need to enclose it in quotes.

Some examples:

- `m*`: Matches any object whose name begins with "m".

- `m* AND *-{0..250}`: Matches any object whose name begins with "m" and ends with a dash and a number between 0 and 250.
- `NOT *timer*`: Matches any object whose name does not contain the substring "timer".
- `NOT (*timer* OR *timeout*)`: Matches any object whose name contains neither "timer" nor "timeout".
- `kind =~ 3 OR kind =~ {7..9}`: Matches messages with message kind equal to 3, 7, 8, or 9.
- `className =~ IP* AND data-*`: Matches objects whose class name starts with "IP" and name starts with "data-".
- `NOT className =~ omnetpp::cMessage AND byteLength =~ {1500..}`: Matches objects whose class is not cMessage and whose byteLength is at least 1500.
- `"TCP packet" OR "*.packet(15)"`: Quotation marks are needed when the pattern is a reserved word or contains whitespace or special characters.

There is also a per-module setting that models can adjust programmatically to prevent any animations from happening when inspecting a given module (`setBuiltinAnimationsAllowed()`).

7.7.6 Configuring Fonts

Fig. 7.33: Font selection

The *Fonts* page of the settings dialog allows you to select the typeface and font size for various user interface elements.

7.7.7 The .qtenvrc File

Settings are stored in `.qtenvrc` files. There are two `.qtenvrc` files: one is stored in the current directory and contains project-specific settings, such as the list of open inspectors; the other is saved in the user's home directory and contains global settings.

Note: Inspectors are identified by their object names. If you have several components that share the same name (this is especially common for messages), you may end up with a lot of inspector windows when you start the simulation. In such cases, you can simply delete the `.qtenvrc` file.

7.8 Qtenv and C++

This section describes which C++ API functions various parts of Qtenv use to display data and perform their functions. Most functions are member functions of the `cObject` class.

7.8.1 Inspectors

Inspectors display the hierarchical name (i.e., full path) and class name of the inspected object in the title using the `getFullPath()` and `getClassName()` member functions of `cObject`. The *Go to parent* feature in inspectors uses the `getOwner()` method of `cObject`.

The *Object Navigator* displays the full name and class name of each object (`getFullName()` and `getClassName()`), and also the ID for classes that have one (`getId()` on `cMessage` and `cModule`). When you hover with the mouse, the tooltip displays the info string (`str()` method). The roots of the tree are the network module (`simulation.getSystemModule()`) and the FES (`simulation.getFES()`). Child objects are enumerated with the help of the `forEachChild()` method.

The *Object Inspector* in *Children* mode displays the full name, class name, and info string (`getFullName()`, `getClassName()`, `str()`) of child objects enumerated using `forEachChild()`. `forEachChild()` can only enumerate objects that are subclasses of `cObject`. If you want non-`cObject` variables (e.g., primitive types or STL containers) to appear in the *Children* tree, you need to wrap them into `cObject`. The `WATCH()` macro does exactly that: it creates an object wrapper that displays the variable's value via the wrapper's `str()` method. There are also watch macros for STL containers; they present the wrapped object to Qtenv in a more structured way using custom class descriptors (cClassDescriptor, see below).

One might wonder how the `forEachChild()` method of modules can enumerate messages, queues, and other objects owned by the module. The answer is that the module class maintains a list of owned objects, and `cObject` automatically joins that list.

The *Object Inspector* displays an object's fields by making use of the class descriptor (`cClassDescriptor`) for that class. Class descriptors are automatically generated for new classes by the message compiler. Class descriptors for the OMNEST library classes are also generated by the message compiler; see `src/sim/sim_std.msg` in the source tree.

The *Network Display* uses `cSubmoduleIterator` to enumerate submodules, and its *Go to parent module* function uses `getParentModule()`. Background and submodule rendering is based on display strings (`getDisplayString()` method of `cComponent`).

The module log page of *Log Viewer* displays the output to `EV` streams from modules and channels.

The message/packet traffic page of *Log Viewer* shows information based on stored copies of sent messages (the copy is created using `dup()`) and stored sendhop information. The

Name column displays the message name (`getFullName()`). However, the *Info* column does not display the string returned from `str()`, but instead, it displays strings produced by a `cMessagePrinter` object. Message printers can be dynamically registered.

7.8.2 During Simulation

Qtenv sets up a network by calling `simulation.setupNetwork()`, then immediately proceeds to invoke `callInitialize()` on the root module. During simulation, `simulation.takeNextEvent()` and `simulation.executeEvent()` are called iteratively. When the simulation ends, Qtenv invokes `callFinish()` on the root module; the same happens when you select the *Conclude Simulation* menu item. The purpose of `callFinish()` is to record summary statistics at the end of a successful simulation run, so it will be skipped if an error occurs during simulation. On exit, and before a new network is set up, `simulation.deleteNetwork()` is called.

The *Debug Next Event* menu item issues the `int3` x86 assembly instruction on Windows and raises a `SIGTRAP` signal on other systems.

7.9 Reference

7.9.1 Command-Line Options

A simulation program built with Qtenv accepts the following command-line switches:

- `-h`: The program prints a help message and exits.
- `-u Qtenv`: Causes the program to start with Qtenv (this is the default, unless the program hasn't been linked with Qtenv or has another custom environment library with a higher priority than Qtenv).
- `-f filename`: Specifies the name of the configuration file. The default is `omnetpp.ini`. Multiple `-f` switches can be given; this allows partitioning your configuration file. For example, one file can contain your general settings, another one can contain most of the module parameters, and a third one can contain the module parameters you frequently change. The `-f` switch is optional and can be omitted.
- `-l filename`: Loads a shared library (`.so` file on Unix, `.dll` on Windows, and `.dylib` on Mac OS X). Multiple `-l` switches are accepted. Shared libraries may contain simple modules and other arbitrary code. File names may be specified without the file extension and the `lib` name prefix (i.e., `foo` instead of `libfoo.so`).
- `-n filepath`: When present, overrides the `NEDPATH` environment variable and sets the source locations for simulation NED files.
- `-c configname`: Selects an INI configuration for execution.
- `-r runnumber`: Takes the same effect as (but takes priority over) the *qtenv-default-run*= INI file configuration option. Run filters are also accepted. If there is more than one matching run, they are grouped at the top of the combobox.

7.9.2 Environment Variables

- `OMNETPP_IMAGE_PATH`: Controls where Qtenv loads images for network graphics (modules, background, etc.) from. The value should be a semicolon-separated list of directories, but on non-Windows systems, the colon is also accepted as a separator. The default is `./bitmaps;./images;<omnetpp>/images`, which means that Qtenv looks into the `bitmaps` and `images` folders of the simulation, as well as the `images` folder in your installation's working directory. The directories will be scanned recursively, and subdirectory names become part of the icon name. For example, if an `images/` directory is listed, the file `images/misc/foo.png` will be registered as an icon named `misc/foo`. Qtenv accepts PNG, JPG, and GIF files.
- `OMNETPP_DEBUGGER_COMMAND`: When set, it overrides the factory default for the command used to launch the just-in-time debugger (`debugger-attach-command`). It must contain '%u' (which will be substituted with the process ID of the simulation), and should not contain any additional '%' characters. Since the command has to return immediately, on Linux and macOS, it is recommended to end it with an ampersand ('&'). The settings on the command line or in an `.ini` file take precedence over this environment variable.

7.9.3 Configuration Options

Qtenv accepts the following configuration options in the INI file.

- `qtenv-extra-stack`: Specifies the extra amount of stack (in kilobytes) reserved for each `activity()` simple module when the simulation is run under Qtenv. This value is significantly higher than the similar one for Cmdenv (handling GUI events requires a large amount of stack space).
- `qtenv-default-config`: Specifies which INI file configuration Qtenv should automatically set up after startup. If there is no such option, Qtenv will ask which configuration to set up.
- `qtenv-default-run`: Specifies which run of the selected configuration Qtenv should set up after startup. If there is no such option, Qtenv will ask.

All other Qtenv settings can be changed via the GUI and are saved into the `.qtenvrc` file in the user's home directory or in the current directory.

CHAPTER

EIGHT

SEQUENCE CHARTS

8.1 Introduction

This chapter describes the Sequence Chart and the Eventlog Table tools. Both of them display an eventlog file recorded by the OMNEST simulation kernel.

An eventlog file contains a log of messages sent during the simulation and the details of events that prompted their sending or reception. This includes both messages sent between modules and self-messages (timers). The user can control the amount of data recorded from messages, start/stop time, which modules to include in the log, and so on. The file also contains the topology of the model (i.e. the modules and their interconnections).

Note: Please refer to the OMNEST Manual for further details on eventlog files and their exact format.

The Sequence Chart displays eventlog files in a graphical form, focusing on the causes and consequences of events and message sends. They help the user understand complex simulation models and assist with the correct implementation of the desired component behaviors. The Eventlog Table displays an eventlog file in a more detailed and direct way. It is in a tabular format, so that it can show the exact data. Both tools can display filtered eventlogs created via the Eventlog Tool filter command as described in the OMNEST Manual, by a third-party custom filter tool, or by the IDE's in-memory filtering.

Using these tools, you will be able to easily examine every detail of your simulation back and forth in terms of simulation time or events. You will be able to focus on the behavior instead of the statistical results of your model.

8.2 Creating an Eventlog File

The INI File Editor in the OMNEST IDE provides a group of widgets in the *Output Files* section to configure automatic eventlog recording. To enable it, simply put a checkmark next to its checkbox, or insert the line

```
record-eventlog = true
```

into the INI file. Additionally, you can use the `--record-eventlog` command-line option or just click the record button on the Qtenv toolbar before starting the simulation.

Fig. 8.1: INI file eventlog configuration

By default, the recorded eventlog file will be put in the project's `results` directory, with the name `$configname-$runnumber.elog` .

Warning: If you override the default file name, please make sure that the file extension is `elog` , so that the OMNEST IDE tools will be able to recognize it automatically.

The 'recording intervals' and 'record events' configuration keys control which events will be recorded based on their simulation time and on the module where they occur. The 'message details' configuration key specifies what will be recorded from a message's content. Message content will be recorded each time a message gets sent.

The amount of data recorded will affect the eventlog file size, as well as the execution speed of the simulation. Therefore, it is often a good idea to tailor these settings to get a reasonable tradeoff between performance and details.

Note: Please refer to the OMNEST Manual for a complete description of eventlog recording settings.

8.3 Sequence Chart

This section describes the Sequence Chart in detail, focusing on its features without a particular example.

The Sequence Chart is divided into three parts: the top gutter, the bottom gutter, and the main area. The gutters show the simulation time while the main area displays module axes, events, and message sends. The chart grows horizontally with simulation time and vertically with the number of modules. Module axes can optionally display enumerated or numerical vector data.

There are various options that control how and what the Sequence Chart displays. Some of these are available on the toolbar, while others are accessible only from the context menu.

8.3.1 Legend

Graphical elements on the Sequence Chart represent modules, events, and messages, as listed in the following table.

	simple module axis
	compound module axis
	axis with attached vector data
	module full path as axis label
	(hollow circle) initialization event
	(green disk) self-message processing event
	(red disk) message processing event
	event number
	(blue arrow, arched) self-message
	(blue arrow) message send
	(green dotted arrow) message reuse
	(brown dotted arrow) method call
	(arrow with a dashed segment) message send that goes far away; split arrow
	(arrow with zigzag) virtual message send; zigzag arrow
	(blue parallelogram) transmission duration; reception at start
	(blue parallelogram) transmission duration; reception at end
	(blue strips) split transmission duration; reception at start
	(blue strips) split transmission duration; reception at end
	(blue letters) message name
	(brown letters) method name
	(gray background) zero simulation time region
	(dashed gray line) simulation time hairline

8.3.2 Timeline

Simulation time may be mapped onto the horizontal axis in various ways; linear mapping is only one of the ways. The reason for having multiple mapping modes is that intervals between interesting events are often of different magnitudes (e.g. microsecond timings in a MAC protocol versus multi-second timeouts in higher layers), which is impossible to visualize using a linear scale.

The available timeline modes are:

- Linear – the simulation time is proportional to the distance measured in pixels.
- Event number – the event number is proportional to the distance measured in pixels.
- Step – the distance between subsequent events, even if they have non-subsequent event numbers, is the same.
- Nonlinear – the distance between subsequent events is a nonlinear function of the simulation time between them. This makes the figure compact even if there are several

magnitudes difference between simulation time intervals. On the other hand, it is still possible to decide which interval is longer and which one is shorter.

- Custom nonlinear – like nonlinear. This is useful in those rare cases when the automatic nonlinear mode does not work well. The best practice is to switch to *Nonlinear* mode first and then to *Custom nonlinear*, so that the chart will continuously refresh as the parameters change. At the extreme, you can set the parameters so that the nonlinear mode becomes equivalent to linear mode or step mode.

You can switch between timeline modes using the button on the toolbar or from the context menu.

8.3.3 Zero Simulation Time Regions

It is quite common in simulation models for multiple events to occur at the same simulation time, possibly in different modules. A region with a gray background indicates that the simulation time does not change along the horizontal axis within the area, thus all events inside it have the same simulation time associated with them.

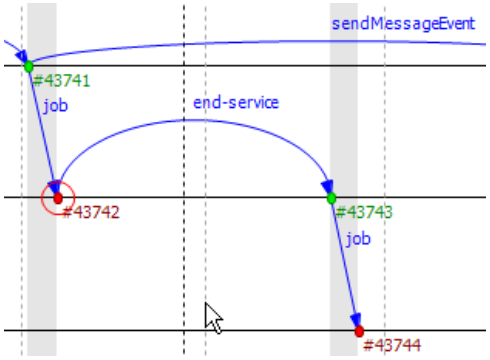


Fig. 8.2: Nonlinear simulation time

8.3.4 Module Axes

The Sequence Chart's vertical axis corresponds to modules in the simulation. By default, each simple module is displayed on a separate horizontal axis, and events that occurred in that module are shown as circles on it. A compound module is represented with a double line, and it will display events from all contained simple modules, except internal events and those that have their own axes displayed. An event is internal to a compound module if it only processes a message from, and sends out messages to, other modules inside.

It is not uncommon for some axes to not have any events at all. These axes would waste space by occupying some place on the screen, so by default, they are omitted from the chart unless the *Show Axes Without Events* option is turned on. The discovery process is done lazily as you navigate through the chart, and it may add new axes dynamically as soon as it turns out that they actually have events.

Module axes can be reordered with the option *Axis Ordering Mode* . Ordering can be manual or sorted by module name, module ID, or by minimizing the total number of axes that arrows cross.

Note: The algorithm that minimizes crossings works by taking a random sample from the file and determines the order of axes from that (which means that the resulting order will only be an approximation). A more precise algorithm, which takes all arrows into account, would not be practical because of the typically large size of eventlog files.

8.3.5 Gutter

The upper and lower edges of the Sequence Chart show a gutter that displays the simulation time. The left side of the top gutter displays a *time prefix* value, which should be added to each individual simulation time shown at the vertical hairlines. This reduces the number of characters on the gutter and allows easier recognition of simulation time changes in the significant digits. The right side of the figure displays the simulation time range that is currently visible within the window.

Tip: To see the simulation time at a specific point on the chart, move the mouse to the desired place and read the value in the blue box horizontally aligned with the mouse on the gutter.

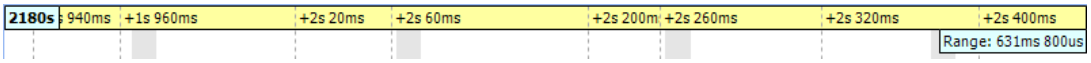


Fig. 8.3: Gutter and range

8.3.6 Events

Events are displayed as filled circles along the module axes. A green circle represents the processing of a self-message, while a red circle is an event caused by receiving a message from another module. The event with event number zero represents the module initialization phase and may spread across multiple module axes because the simulation kernel calls each module during initialization. This event is displayed with a white background.

Event numbers are displayed below and to the right of their corresponding events and are prefixed with '#'. Their colors change according to their events' colors.

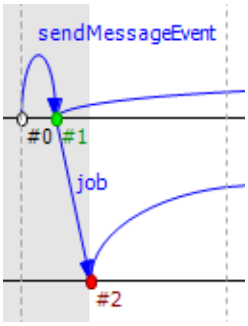


Fig. 8.4: Various event types

8.3.7 Messages

The Sequence Chart represents message sends with blue arrows. Vertically, the arrow starts at the module that sent the message and ends at the module that processed the message. Horizontally, the start and end points of the arrow correspond to the sender and receiver events. The message's name is displayed near the middle of the arrow, but not exactly in the middle to avoid overlapping with other names between the same modules.

Sometimes, when a message arrives at a module, it simply stores it and later sends the exact same message out. The events where the message arrived and where the message was actually sent are in a so-called "message reuse" relationship. This is represented by a green dotted arrow. These arrows are not shown by default because timer self-messages are usually reused continuously. Showing these arrows would add unnecessary complexity to the chart

and make it hard to understand. To show and hide these arrows, use the button *Show Reuse Messages* on the toolbar.

Sometimes, depending on the zoom factor, a message send goes far away on the chart. In this case, the line is split into two smaller parts that are displayed at the two ends pointing towards each other, but without a continuous line connecting them. At one end of both arrow pieces is a dotted line while at the other end is a solid line. The solid line always points exactly to or from the event to which it is connected. The other line, which is dotted, either specifies the module where the arrow starts or ends, or in the case of a self-message, it points toward the other arrow horizontally.

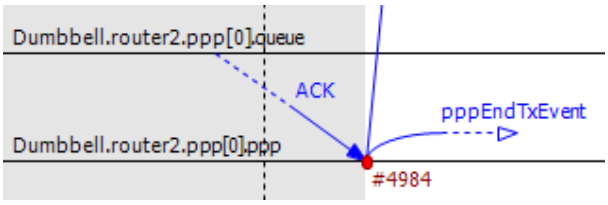


Fig. 8.5: Split arrows

8.3.8 Displaying Module State on Axes

It is possible to display module state on an axis. The axis is then rendered as a colored strip that changes color every time the module state changes. The data is taken from an output vector in an *output vector file*, normally recorded by the simulation together with the eventlog file.

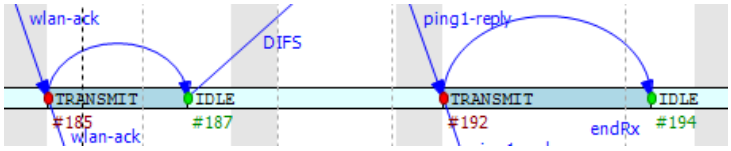


Fig. 8.6: Axis with state information displayed

To attach an output vector to an axis, right-click the desired axis and select *Attach Vector to Axis* from the context menu. You will be prompted for an output vector file and for a vector in the file. If the vector is of type enum (that is, it has metadata attached that assigns symbolic names to values, e.g., `IDLE` for 0, `TRANSMIT` for 1, etc.), then the chart will display symbolic names inside the strip; otherwise, it will display the values as numbers. The background coloring for the strip is automatic.

Note: Recording output vectors is explained in the *OMNEST Simulation Manual*. It is recommended to turn on recording event numbers (`**.vector-record-eventnumbers = true` ini file setting), because that allows the Sequence Chart tool to display state changes accurately even if there are multiple events at the same simulation time.

The format of output vector files is documented in an appendix of the Manual. To see whether a given output vector is suitable for the Sequence Chart, search for the vector declaration (`vector... `` line) in the file. When event numbers are enabled, the vector declaration will end in ``ETV` (not `TV`). If a vector has an enum attached, there will be an `attr enum` line after the vector declaration. An example vector declaration with an enum:

8.3.9 Zooming

To zoom in or out horizontally along the timeline, use the *Zoom In* and *Zoom Out* buttons on the toolbar. To decrease or increase the distance between the axes, use the *Increase/Decrease Spacing* commands.

Warning: When you zoom out, more events and messages become visible on the chart, making it slower. When you zoom in, message lines start breaking, making it less informative. Try to keep a reasonable zoom level.

8.3.10 Navigation

To scroll through the Sequence Chart, use either the scroll bars, drag with the left mouse button, or scroll with the mouse wheel using the `Shift` modifier key for horizontal scroll.

There are also navigation options to go to the previous `Shift+LEFT` or next `Shift+RIGHT` event in the same module.

Similar to navigating in the Eventlog Table, to go to the cause event, press `Ctrl+LEFT`. To go to the arrival of a message send, press `Ctrl+RIGHT` while an event is selected.

8.3.11 Tooltips

The Sequence Chart displays tooltips for axes, events, message sends, and reuses. When a tooltip is shown for any of the above, the chart will highlight the corresponding parts. Sometimes, when the chart is zoomed out, it might show a complex tooltip immediately because there are multiple items under the mouse.

Tip: To measure the simulation time difference between two events, select one of them while staying at the other to display the tooltip.

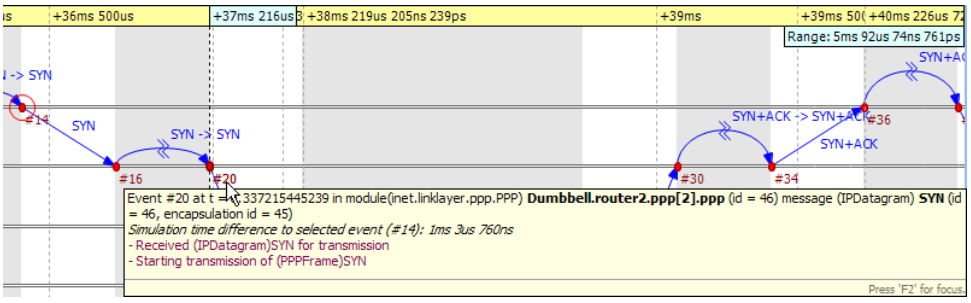


Fig. 8.7: Event tooltip

8.3.12 Bookmarks

Just like the Eventlog Table, the Sequence Chart also supports bookmarks to make navigation easier. Bookmarks are saved for the files rather than the various editors; therefore, they are shared between them. The chart highlights bookmarked events with a circle around them, similar to primary selection but with a different color.

8.3.13 Exporting

The Sequence Chart supports exporting continuous parts into SVG format for documentation purposes. This function is available from the context menu . You can export the whole Sequence Chart, a region between two selected events, or the currently visible area.

8.3.14 Associated Views

When you open an eventlog file in the Sequence Chart editor, it will automatically open the *Eventlog Table View* with the same file. If you select an event on the Sequence Chart editor, then the *Eventlog Table View* will jump to the same event, and vice versa. This interconnection makes navigation easier, and you can immediately see the details of the selected event's raw data.

8.3.15 Filtering

You can also filter the contents of the Sequence Chart. This actually means that some of the events are not displayed on the chart, so that the user can focus on the relevant parts. When filtering is turned on (displayed in the status line), some of the message arrows might have a filter sign (a double zigzag crossing the arrow line's center). Such a message arrow means that there is a message going out from the source module, which after processing in some other filtered-out modules, reaches the target module. The message name of the arrow in this case corresponds to the first and the last message in the chain that was filtered out.

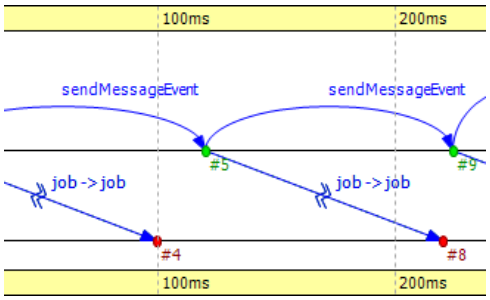


Fig. 8.8: Zigzag arrows

When a module filter is used, it will determine which modules will have axes. If the events that occurred in a module are completely filtered out, then the Sequence Chart will not display the superfluous axis belonging to that module. This reduces the number of axes and makes it easier to understand the figure.

Events may not have subsequent event numbers, which means that the events in between have been filtered out. At the extreme, the chart may even be empty, meaning that there are no matching events at all.

To filter the Sequence Chart, open the *Filter Dialog* using the filter button on the toolbar. You can also filter from the context menu using the shortcuts provided for events and message sends currently under the mouse.

8.4 Eventlog Table

This section describes the Eventlog Table in detail, focusing on its features without a particular example.

The Eventlog Table has one row per line in the eventlog file. It has three columns. The first two are called event number and simulation time respectively. They show the values corresponding to the simulation event where the line was recorded. The third column, called details, contains the actual data, which varies for each line kind. The different kinds of lines can be easily recognized by their icons. Some lines, such as sending a message through a sequence of gates, relate to each other and are indented so that the user can recognize them more easily.

There are various options that control how and what the Eventlog Table displays. Some of these are available on the toolbar, while others are accessible only from the context menu.

8.4.1 Display Mode

The eventlog file content may be displayed in two different notations. The *Raw* data notation shows exactly what is present in the file.

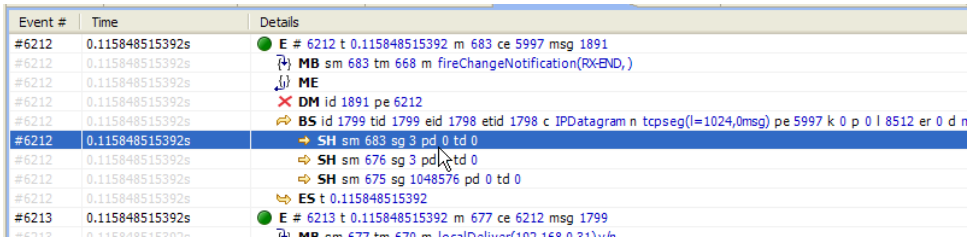


Fig. 8.9: Raw notation

The *Descriptive* notation, after some preprocessing, displays the log file in a readable format. It also resolves references and types so that less navigation is required to understand what is going on. To switch between the two, use the *Display Mode* button on the toolbar or the context menu.

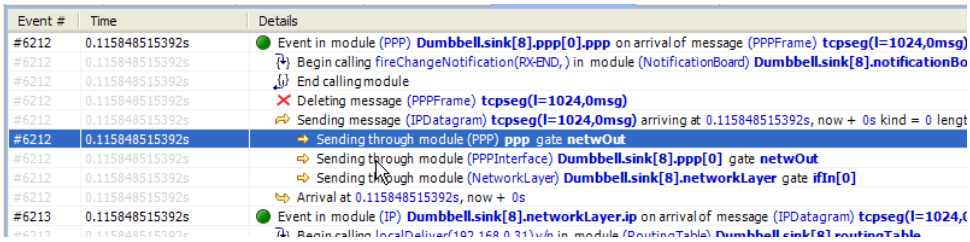


Fig. 8.10: Descriptive notation

8.4.2 Name Mode

There are three different ways to display names in the Eventlog Table; it is configurable with the *Name Mode* option. Full path and full name show what you would expect. The smart mode uses the context of the line to decide whether a full path or a full name should be displayed. For each event line, this mode always displays the full path. For all other lines, if the name is the same as the enclosing event's module name, then it shows the full name only. This choice makes lines shorter and allows for faster reading.

8.4.3 Type Mode

The option called *Type Mode* can be used to switch between displaying the C++ class name or the NED type name in parentheses before module names. This is rarely used, so it is only available from the context menu.

8.4.4 Line Filter

The Eventlog Table may be filtered by using the *Line Filter* button on the toolbar. This option allows filtering for lines with specific kinds. There are some predefined filters.

You can also provide a custom filter pattern, referring to fields present in *Raw* mode, using a match expression. The following example is a custom filter that will show message sends where the message's class is AirFrame.

```
BS and c(AirFrame)
```

Please refer to the OMNEST Manual for more details on match expressions.

Note: To avoid confusion, event lines marked with green circles are always shown in the Eventlog Table and are independent of the line filter.

8.4.5 Navigation

You can navigate using your keyboard and mouse just like in any other table. There are a couple of non-standard navigation options in the context menu, which can also be used with the keyboard.

The simplest are the *Goto Event* and the *Goto Simulation Time*, both of which simply jump to the designated location.

There are navigation options for going to the previous `Alt+UP` or next `Alt+DOWN` event in general, and to go to the previous `Shift+UP` or next `Shift+DOWN` event in the same module.

Some of the navigation options focus on the causes of events and consequences of message sends. To go to the cause event, press `Ctrl+UP`. To go to the arrival of a message send, press `Ctrl+DOWN`, after selecting the message being sent.

Finally, there are navigation options for message reuse relationships. You can go to the original event of a message from the line where it was being reused. In the other direction, you can go to the reused event of a message from the event where it was received. These options are enabled only if they actually make sense for the current selection.

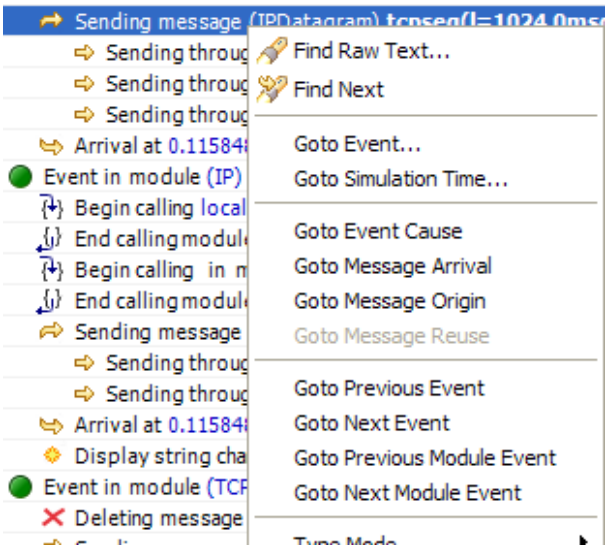


Fig. 8.11: Navigation context menu

8.4.6 Selection

The Eventlog Table uses multiple selection even though most of the user commands require single selection.

8.4.7 Searching

For performance reasons, the search function works directly on the eventlog file and not the text displayed in the Eventlog Table. It means that some static text present in *Descriptive* mode cannot be found. Usually, it is easier to figure out what to search for in *Raw* mode, where the eventlog file's content is directly displayed. The search can work in both directions, starting from the current selection, and may be case insensitive. To repeat the last search, use the *Find Next* command.

8.4.8 Bookmarks

For easier navigation, the Eventlog Table supports navigation history. This is accessible from the standard IDE toolbar just like for other kinds of editors. It works by remembering each position where the user stayed more than 3 seconds. The navigation history is temporary and thus it is not saved when the file is closed.

Persistent bookmarks are also supported, and they can be added from the context menu. A Bookmarked event is highlighted with a different background color.

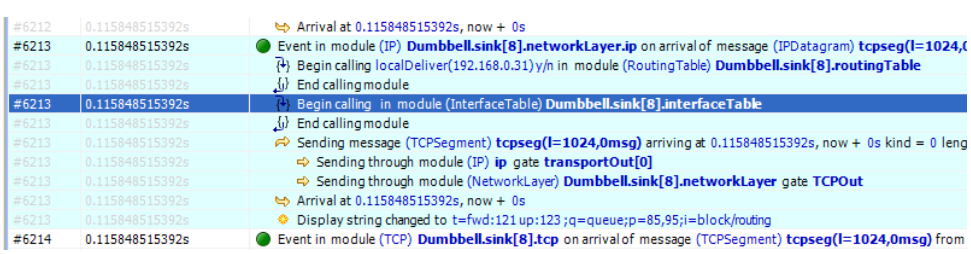


Fig. 8.12: A bookmark

To jump to a bookmark, use the standard *Bookmark View* (this is possible even after restarting the IDE).

8.4.9 Tooltips

Currently, only the message send lines have tooltips. If message detail recording was configured for the simulation, then a tooltip will show the recorded content of a message send over the corresponding line.

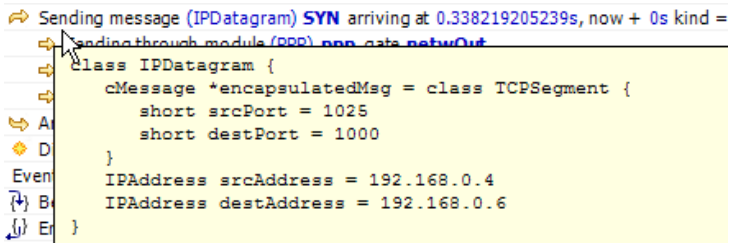


Fig. 8.13: A message send tooltip

8.4.10 Associated Views

When you open an eventlog file in the Eventlog Table editor, it will automatically open the *Sequence Chart View* with the same file. If you select an event on the Eventlog Table editor, then the *Sequence Chart View* will jump to the same event, and vice versa. This interconnection makes navigation easier, and you can immediately see the cause and effect relationships of the selected event.

8.4.11 Filtering

If the Eventlog Table displays a filtered eventlog, then subsequent events may not have subsequent event numbers. This means that the events in between have been filtered out. At the extreme, the table may even be empty, which means that there are no matching events at all.

8.5 Filter Dialog

The content of an eventlog can be filtered within the OMNEST IDE. This is on-the-fly filtering as opposed to the file content filtering provided by the *Eventlog* tool. To use on-the-fly filtering, open the filter configuration dialog with the button on the toolbar, enable some of the range, module, message, or trace filters, set the various filter parameters, and apply the settings. The result is another eventlog, resident in memory, that filters out some events.

Note: Similar to the command line **`opp_eventlogtool`** described in the OMNEST Manual, the in-memory filtering can only filter out whole events.

In-memory, on-the-fly filtering means that the filter's result is not saved into an eventlog file, but it is computed and stored within memory. This allows rapid switching between different views of the same eventlog within both the *Sequence Chart* and the *Eventlog Table*.

The filter configuration dialog shown in Fig. 8.14 has many options. They are organized into a tree with each part restricting the eventlog's content. The individual filter components may be turned on and off independent of each other. This allows remembering the filter settings even if some of them are temporarily unused.

The combination of various filter options might be complicated and hard to understand. To make it easier, the *Filter Dialog* automatically displays the current filter in a human-readable form at the bottom of the dialog.

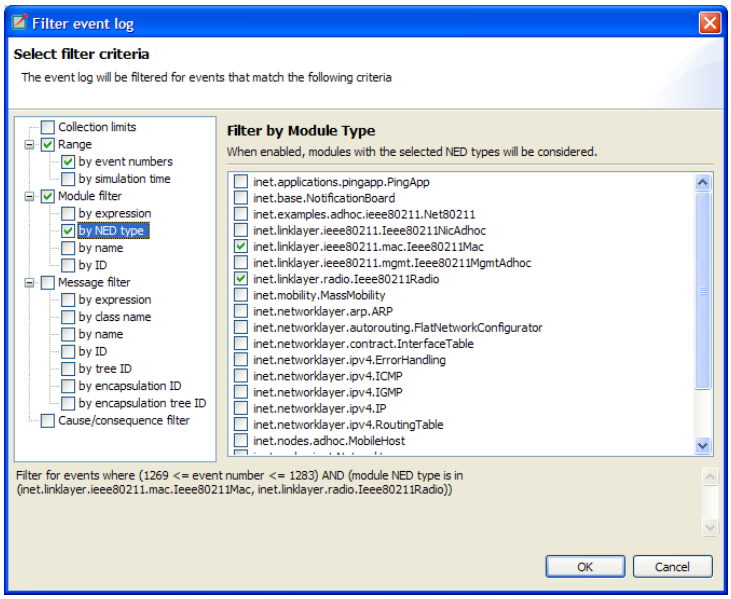


Fig. 8.14: Filter Dialog

8.5.1 Range Filter

This is the simplest filter, which filters out events from the beginning and end of the eventlog. It might help to reduce the computation time dramatically when defining filters that would otherwise be very expensive to compute for the whole eventlog file.

8.5.2 Module Filter

With this kind of filter, you can filter out events that did not occur in any of the specified modules. The modules that will be included in the result can be selected by their NED type, full path, module ID, or by a match expression. The expression may refer to the raw data present in the lines marked with 'MC' in the eventlog file.

8.5.3 Message Filter

This filter is the most complicated one. It allows filtering for events that either process or send specific messages. The messages can be selected based on their C++ class name, message name, various message IDs, and a match expression. The expression may refer to the raw data present in the lines marked with 'BS' in the eventlog file.

There are four different message IDs to filter, each with different characteristics. The most basic one is the ID, which is unique for each constructed message, independent of how it was created. The tree ID is special because it gets copied over when a message is created by copying (duplicating) another. The encapsulation ID is different in that it gives the ID of the innermost encapsulated message. Finally, the encapsulation tree ID combines the two by providing the innermost encapsulated message's tree ID.

8.5.4 Tracing Causes/Consequences

The trace filter allows filtering for causes and consequences of a particular event specified by its event number. The cause/consequence relation between two events means that there is a message send/reuse path from the cause event to the consequence event. If there was a message reuse in the path, then the whole path is considered to be a message reuse itself.

Warning: Since computing the causes and consequences in an eventlog file that is far away from the traced event might be a time-consuming task, the user can set extra range limits around the traced event. These limits are separate from the range filter due to being relative to the traced event. This means that if you change the traced event, there is no need to change the range parameters. It is strongly recommended that users provide these limits when tracing events to avoid long-running operations.

8.5.5 Collection Limits

When an in-memory filter is applied to an eventlog, it does not only filter out events, but it also provides automatic discovery for virtual message sends. It means that two events far away and not directly related to each other might have a virtual message send (or reuse) between them. Recall that there is a virtual message send (or reuse) between two events if and only if there is a path of message sends (or reuses) connecting the two.

The process of collecting these virtual message dependencies is time-consuming and thus has to be limited. There are two options. The first one limits the number of virtual message sends collected per event. The other one limits the depth of cause/consequence chains during collection.

8.5.6 Long-Running Operations

Sometimes, computing the filter's result takes a lot of time, especially when tracing causes/consequences without specifying proper range limits in terms of event numbers or simulation times. If you cancel a long-running operation, you can go back to the *Filter Dialog* to modify the filter parameters, or simply turn the filter off. To restart drawing, use the refresh button  on the toolbar.

Tip: Providing a proper range filter is always a good idea to speed up computing the filter's result.

8.6 Other Features

Both the Sequence Chart and the Eventlog Table tools can be used as editors and also as views. The difference between an editor or a view is quite important because there is only at most one instance of a view of the same kind. It means that even if multiple eventlog files are open in Sequence Chart editors, there is no more than one *Eventlog Table* view shared between them. This single view will automatically display the eventlog file of the active editor. It will also remember its position and state when it switches among editors. For more details on editors and views, and their differences, please refer to the Eclipse documentation.

Note: Despite the name "editor," which is a concept of the Eclipse platform, neither the *Sequence Chart*, nor the *Eventlog Table* can be used to actually change the contents of an eventlog file.

It is possible to open the same eventlog file in multiple editors and to navigate to different locations or use different display modes or filters in a location. Once an eventlog is open in an editor, you can use the *Window → New Editor* to open it again.

Tip: Dragging one of the editors from the tabbed pane to the side of the editor's area allows you to interact with both simultaneously.

8.6.1 Settings

There are various settings for both tools that affect the display, such as display modes, content position, filter parameters, etc. These user-specified settings are automatically saved for each file and are reused whenever the file is revisited. The per-file settings are stored under the OMNEST workspace in the directory `.metadata.pluginsorg.eclipse.core.resources.projects<project-name>`.

8.6.2 Large File Support

Since an eventlog file might be several gigabytes, both tools are designed in a way that allows for efficient displaying of such a file without requiring large amounts of physical memory to load it at once. As you navigate through the file, physical memory is filled up with the content lazily. Since it is difficult to reliably identify when the system is getting low on physical memory, it is up to the user to release the allocated memory when needed. This operation, although usually not required, is available from the context menu as *Release Memory*. It does not affect the user interface in any way.

The fact that the eventlog file is loaded lazily and optionally filtered also means that the exact number of lines and events it contains cannot be easily determined. This affects the way scrollbars work in the lazy directions: horizontal for the Sequence Chart and vertical for the Eventlog Table. These scrollbars act as a non-linear approximation in that direction. For the most part, the user will be unaware of these approximations unless the file is really small.

8.6.3 Viewing a Running Simulation's Results

Even though the simulation kernel keeps the eventlog file open for writing while the simulation is running, it may also be open in the OMNEST IDE simultaneously. Both tools can be guided by pressing the END key to follow the eventlog's end as new content is appended to it. If you pause the simulation in the runtime environment, then after a few seconds the tools will refresh their content and jump to the very end. This process makes it possible to follow the simulation step-by-step on the Sequence Chart.

8.6.4 Caveats

Sometimes, drawing the Sequence Chart may take a lot of time. Zooming out too much, for example, might result in slow response times. A dialog might pop up telling the user that a long-running eventlog operation is in progress. You can safely cancel these operations at any time you like or just wait until they finish. To restart the rendering process, simply press the refresh button on the toolbar. Before refreshing, it is a good idea to revert to some defaults (e.g. default zoom level) or revert the last changes (e.g. navigate back, turn filter off, etc.).

Warning: An operation that runs for an unreasonably long time might be a sign of a problem that should be reported.

8.7 Examples

This section will guide you through the use of the Sequence Chart and Eventlog Table tools, using example simulations from OMNEST and the INET Framework. Before running any of the simulations, make sure that eventlog recording is enabled by adding the line

```
record-eventlog = true
```

to the `omnetpp.ini` file in the simulation's directory. To open the generated eventlog in the OMNEST IDE, go to the example's `results` directory in the *Resource Navigator* View and double-click the log file. By default, the file will open in the Sequence Chart.

Tip: To open the file in the Eventlog Table as an editor, right-click the file and choose the corresponding item from the context menu's *Open With* submenu.

8.7.1 Tictoc

The Tictoc example is available in the OMNEST installation under the directory `samples/tictoc`. Tictoc is the most basic example in this chapter and it provides a quick overview of how to use and understand the Sequence Chart.

Start the simulation and choose the simplest configuration, 'Tictoc1,' which specifies only two nodes called 'tic' and 'toc.' During initialization, one of the nodes will send a message to the other. From then on, every time a node receives the message, it will simply send it back. This process continues until you stop the simulation. In Fig. 8.15, you can see how this is represented on a Sequence Chart. The two horizontal black lines correspond to the two nodes and are labeled 'tic' and 'toc.' The red circles represent events, and the blue arrows represent message sends. It is easy to see that all message sends take 100 milliseconds and that the first sender is the node 'tic.'

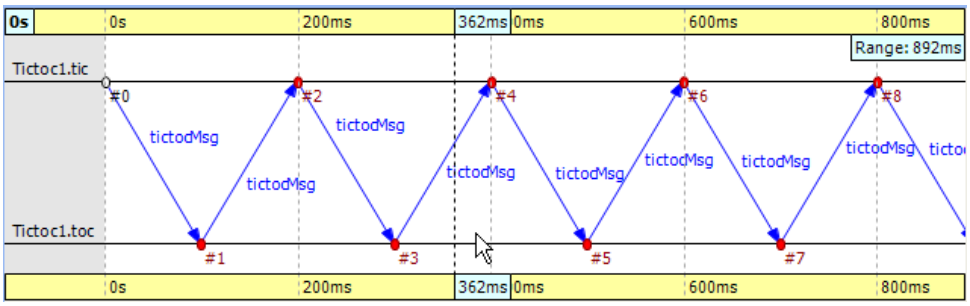


Fig. 8.15: Tictoc with two nodes

In the next Tictoc example, there are six nodes tossing a message around until it reaches its destination. To generate the eventlog file, restart the simulation and choose the configuration 'Tictoc9.' In Fig. 8.16, you can see how the message goes from one node to another, starting from node '0' and passing through it twice more until it finally reaches its destination, node '3.' The chart also shows that this example, unlike the previous one, starts with a self-message instead of immediately sending a message from initialize to another node.

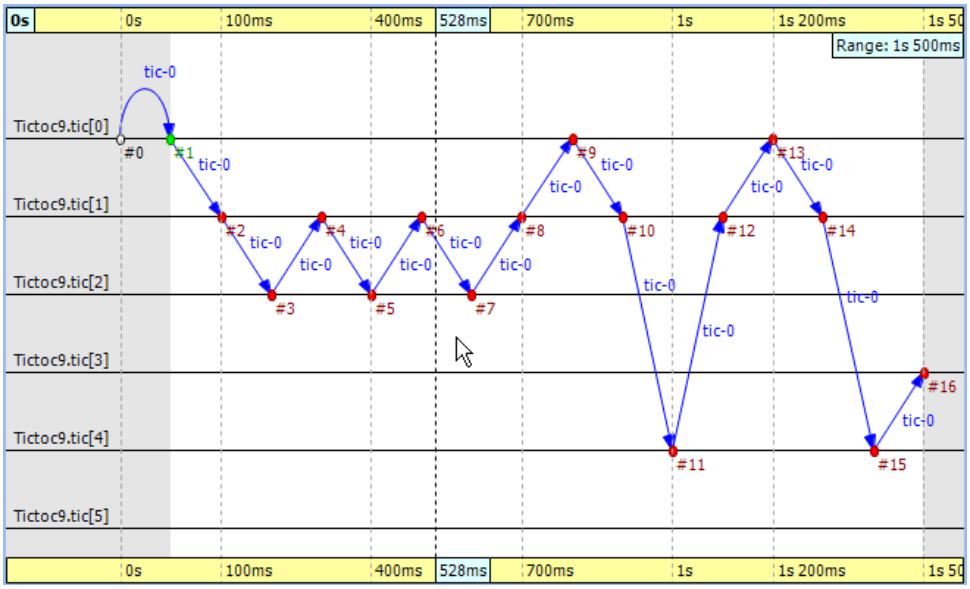


Fig. 8.16: Tictoc with six nodes

Let us demonstrate with this simple example how filtering works with the Sequence Chart. Open the *Filter Dialog* with the toolbar button and put a checkmark for node '0' and '3' on the *Module filter* → *by name* panel, and apply it. The chart now displays only two axes that correspond to the two selected nodes. Note that the arrows on this figure are decorated with zigzags, meaning that they represent a sequence of message sends. Such arrows will be called virtual message sends in the rest of this chapter. The first two arrows show the message returning to node '0' at event #9 and event #13, and the third shows that it reaches the destination at event #16. The events where the message was in between are filtered out.

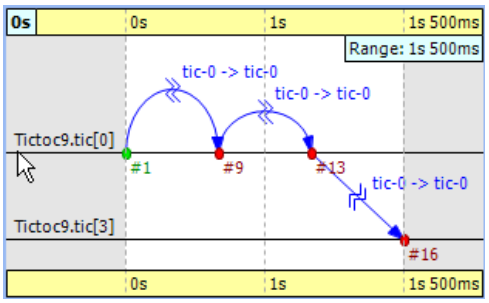


Fig. 8.17: Filtering for nodes '0' and '3'

8.7.2 FIFO

The FIFO example is available in the OMNEST installation under the directory `samples/fifo`. The FIFO is an important example because it uses a queue, which is an essential part of discrete event simulations and introduces the notion of message reuses.

When you start the simulation, choose the configuration 'low job arrival rate' and let it run for a while. In Fig. 8.18, you can see three modules: a `source`, a `queue`, and a `sink`. The simulation starts with a self-message, and then the generator sends the first message to the queue at event #1. It is immediately obvious that the message stays in the queue for a certain period of time, between event #2 and event #3.

Tip: When you select one event and hover with the mouse over the other, the Sequence

Chart will show the length of this time period in a tooltip.

Finally, the message is sent to the `sink`, where it is deleted at event #4.

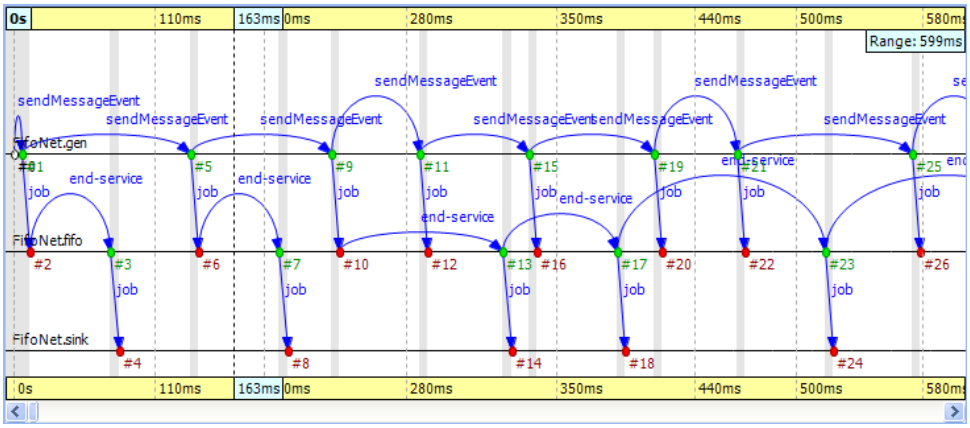


Fig. 8.18: The FIFO example

Something interesting happens at event #12, where the incoming message suddenly disappears. It seems like the queue does not send the message out. Actually, what happens is that the queue enqueues the job because it is busy serving the message received at event #10. Since this queue is a FIFO, it will send out the first message at event #13. To see how this happens, turn on *Show Reuse Messages* from the context menu; the result is shown in Fig. 8.19. It displays a couple of green dotted arrows, one of which starts at event #12 and arrives at event #17. This is a reuse arrow; it means that the message sent out from the queue at event #17 is the same as the one received and enqueued at event #12. Note that the service of this message actually begins at event #13, which is the moment that the queue becomes free after it completes the job received at event #10.

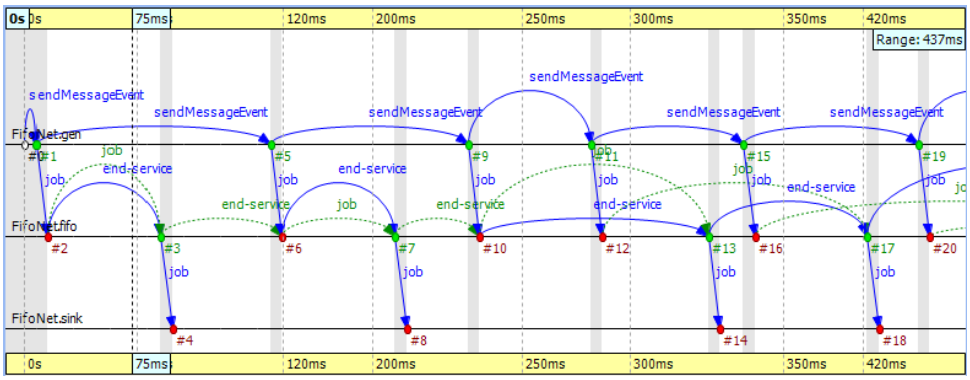


Fig. 8.19: Showing reuse messages

Another type of message reuse is portrayed with the arrow from event #3 to event #6. The arrow shows that the queue reuses the same timer message instead of creating a new one each time.

Note: Whenever you see a reuse arrow, it means that the underlying implementation remembers the message between the two events. It might be stored in a pointer variable, a queue, or some other data structure.

The last part of this example is about filtering out the queue from the chart. Open the *Filter Dialog*, select *sink* and *source* on the *Module filter → by NED type* panel, and apply the change

in settings. If you look at the result in Fig. 8.20, you will see zigzag arrows going from the 'source' to the 'sink.' These arrows show that a message is being sent through the queue from 'source' to 'sink.' The first two arrows do not overlap in simulation time, which means the queue did not have more than one message during that time. The third and fourth arrows do overlap because the fourth job reached the queue while it was busy with the third one. Scrolling forward, you can find other places where the queue becomes empty and the arrows do not overlap.

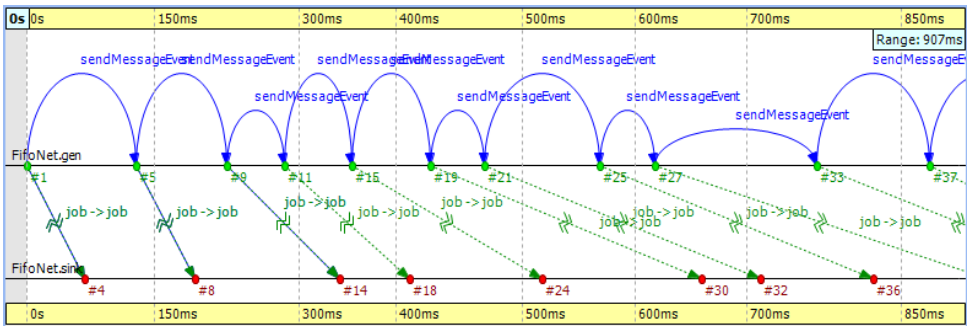


Fig. 8.20: Filtering for the queue

8.7.3 Routing

The Routing example is available in the OMNEST installation under the directory `samples/routing`. The predefined configuration called 'Net10' specifies a network with 10 nodes, with each node having an application, a few queues, and a routing module inside. Three preselected nodes, namely the node '1,' '6,' and '8,' are destinations, while all nodes are message sources. The routing module uses the shortest path algorithm to find the route to the destination. The goal of this example is to create a sequence chart that shows messages traveling simultaneously from multiple sources to their destinations.

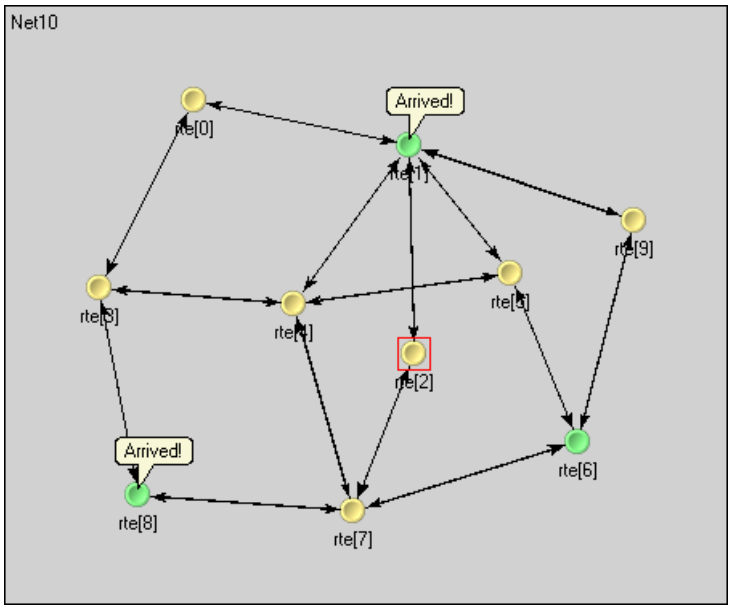


Fig. 8.21: Network with 10 nodes

Since we do not care about the details regarding what happens within nodes, we can simply turn on filtering for the NED type `node.Node`. The chart will have 10 axes, with each axis drawn as two parallel solid black lines close to each other. These are the compound modules that represent the nodes in the network. So far, events could be directly drawn on the simple

module's axis where they occurred, but now they will be drawn on the compound module's axis of their ancestor.

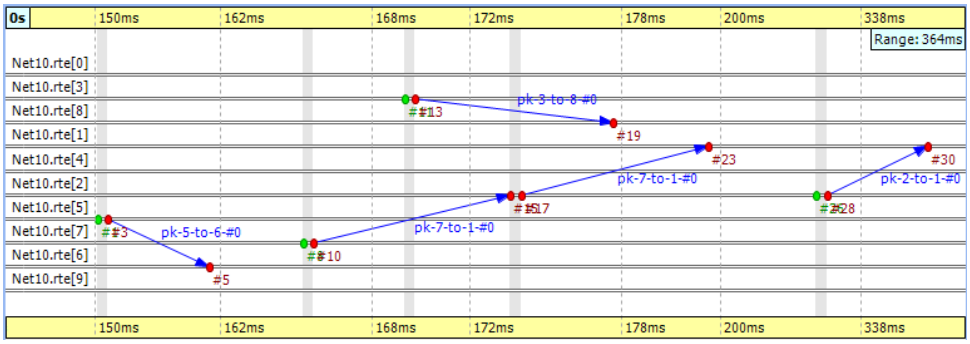


Fig. 8.22: Filtering for the nodes

To reduce clutter, the chart will automatically omit events that are internal to a compound module. An event is internal to a compound module if it only processes a message from, and sends out messages to, other modules inside the compound module.

If you look at Fig. 8.22, you will see a message going from node '7' at event #10 to node '1' at event #23. This message stays in node '2' between event #15 and event #17. The gray background area between them means that zero simulation time has elapsed (i.e., the model does not account for processing time inside the network nodes).

Note: This model contains both finite propagation delay and transmission time; arrows in the sequence chart correspond to the interval between the start of the transmission and the end of the reception.

This example also demonstrates message detail recording configured by

```
eventlog-message-detail-pattern = Packet:declaredOn(Packet)
```

in the INI file. The example in Fig. 8.23 shows the tooltip presented for the second message send between event #17 and event #23.

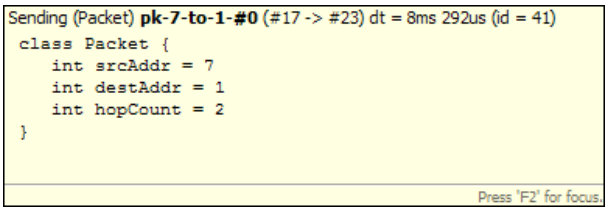


Fig. 8.23: Message detail tooltip

It is very easy to find another message on the chart that goes through the network parallel in simulation time. The one sent from node '3' at event #13 to node '8' arriving at event #19 is such a message.

8.7.4 Wireless

The Wireless example is available in the INET Framework under the directory `examples/adhoc/ieee80211`. The predefined configuration called 'Config1' specifies two mobile hosts moving around on the playground and communicating via the IEEE 802.11 wireless protocol. The network devices are configured for ad-hoc mode, and the transmitter power is set so that hosts can move out of range. One of the hosts is continuously pinging the other.

In this section, we will explore the protocol's MAC layer using two sequence charts. The first chart will show a successful ping message being sent through the wireless channel. The second chart will show ping messages getting lost and being continuously resent.

We would also like to record some message details during the simulation. To perform that function, comment out the following line from `omnetpp.ini`:

```
eventlog-message-detail-pattern = *:(not declaredOn(cMessage) and not␣
↪declaredOn(cNamedObject) and not declaredOn(cObject))
```

To generate the eventlog file, start the simulation environment and choose the configuration 'host1 pinging host0.' Run the simulation in fast mode until event #5000.

Preparing the Result

When you open the *Sequence Chart*, it will show a couple of self-messages named 'move' being scheduled regularly. These are self-messages that control the movement of the hosts on the playground. There is an axis labeled 'pingApp,' which starts with a 'sendPing' message that is processed in an event far away on the chart. This is indicated by a split arrow.

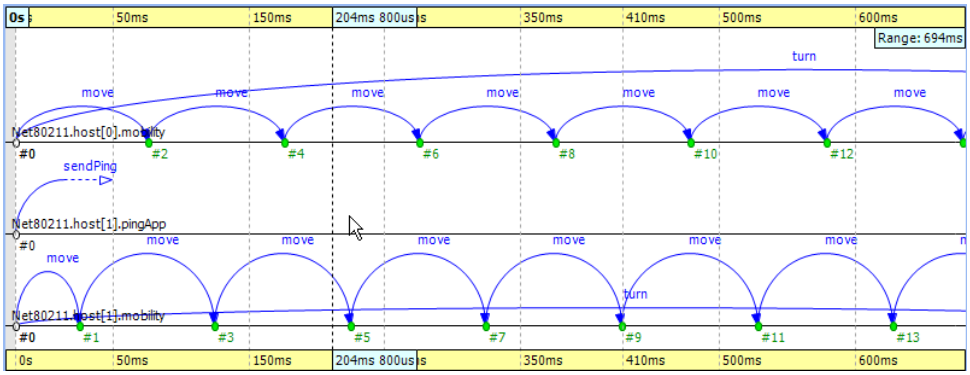


Fig. 8.24: The beginning

You might notice that there are only three axes in Fig. 8.24, even though the simulation model clearly contains more simple modules. This is because the Sequence Chart displays the first few events by default, and in this scenario, they all happen to be within those modules. If you scroll forward or zoom out, new axes will be added automatically as needed.

For this example, ignore the 'move' messages and focus on the MAC layer instead. To begin with, open the *Filter Dialog*, select 'Ieee80211Mac' and 'Ieee80211Radio' on the *Module filter → by NED type* panel, and apply the selected changes. The chart will have four axes, two for the MAC and two for the radio simple modules.

The next step is to attach vector data to these axes. Open the context menu for each axis by clicking on them one by one, and select the *Attach Vector to Axis* submenu. Accept the default vector file offered. Then, choose the vector 'mac:State' for the MAC modules and 'mac:RadioState' for the radio modules. You will have to edit the filter in the vector selection dialog (i.e., delete the last segment) for the radio modules because at the moment, the radio state is recorded by the MAC module, so the default filter will not be right. When this step is

completed, the chart should display four thick colored bars as module axes. The colors and labels on the bars specify the state of the corresponding state machine at the given simulation time.

To aid comprehension, you might want to manually reorder the axes, so that the radio modules are put next to each other. Use the button on the toolbar to switch to manual ordering. With a little zooming and scrolling, you should be able to fit the first message exchange between the two hosts into the window.

Successful Ping

The first message sent by 'host1' is not a ping request but an ARP request. The processing of this message in 'host0' generates the corresponding ARP reply. This is shown by the zigzag arrow between event #85 and event #90. The reply goes back to 'host1,' which then sends a WLAN acknowledge in return. In this process, 'host1' discovers the MAC address of 'host0' based on its IP address.

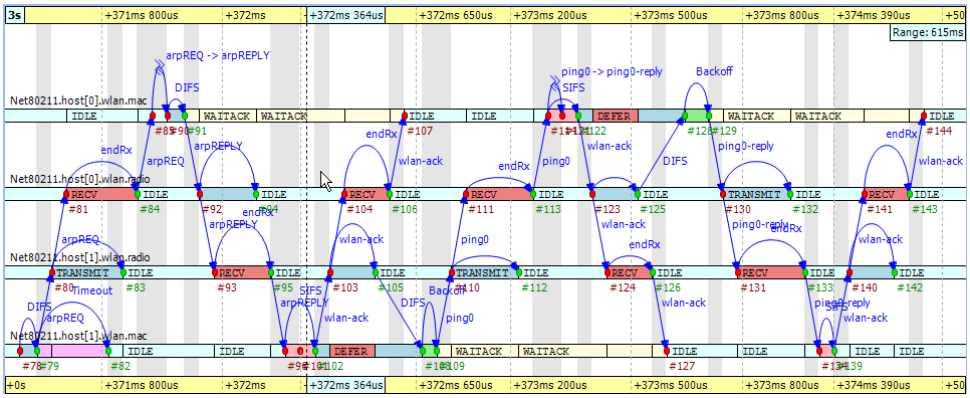


Fig. 8.25: Discovering the MAC address

The send procedure for the first ping message starts at event #105 in 'host1' and finishes by receiving the acknowledge at event #127. The ping reply send procedure starts at event #125 in 'host0' and finishes by receiving the WLAN acknowledge at event #144. If you scroll forward, you can see, as in Fig. 8.26, the second complete successful ping procedure between event #170 and event #206. To focus on the second successful ping message exchange, open the *Filter Dialog* and enter these numbers in the range filter.

Timing is critical in a protocol implementation, so we will take a look at it using the Sequence Chart. The first self message represents the fact that the MAC module listens to the radio for a DIFS period before sending the message out. The message send from event #171 to event #172 occurs in zero simulation time as indicated by the gray background. It represents the moment when the MAC module decides to send the ping request down to its radio module. The backoff procedure was skipped for this message because there was no transmission during the DIFS period. If you look at event #172 and event #173, you will see how the message propagates through the air from 'radio1' to 'radio0.' This finite amount of time is calculated from the physical distance of the two modules and the speed of light. Additionally, by looking at event #172 and event #174, you will notice that the transmission time is not zero. This time interval is calculated from the message's length and the radio module's bitrate.

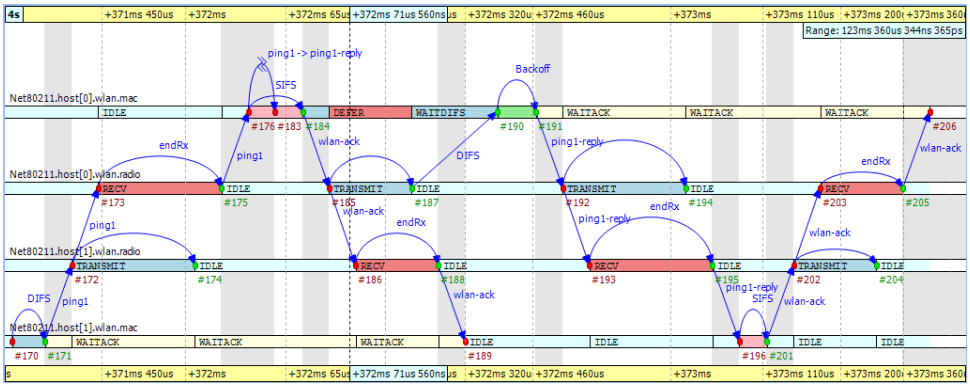


Fig. 8.26: The second ping procedure

Another interesting fact seen in the figure is that the higher-level protocol layers do not add delay for generating the ping reply message in 'host0' between event #176 and event #183. The MAC layer procedure ends with sending back a WLAN acknowledge after waiting a SIFS period.

Finally, you can get a quick overview of the relative timings of the IEEE 802.11 protocol by switching to linear timeline mode. Use the button on the toolbar and notice how the figure changes dramatically. You might need to scroll and zoom in or out to see the details. This shows the usefulness of the nonlinear timeline mode.

You can export this sequence chart for documentation purposes using the context menu's *Export to SVG* option.

Unsuccessful Ping

To see how the chart looks when the ping messages get lost in the air, first turn off range filtering. Then, go to event #1269 by selecting the *Goto Event* option from the *Eventlog Table* View's context menu. In Fig. 8.27, you can see how the receiver radio does not send up the incoming message to its MAC layer due to the signal level being too low. This actually happens at event #1274 in 'host0.' Shortly thereafter, the transmitter MAC layer in 'host1' receives the timeout message at event #1275 and starts the backoff procedure before resending the very same ping message. This process goes on with statistically increasing backoff time intervals until event #1317. Finally, the maximum number of retries is reached, and the message is dropped.

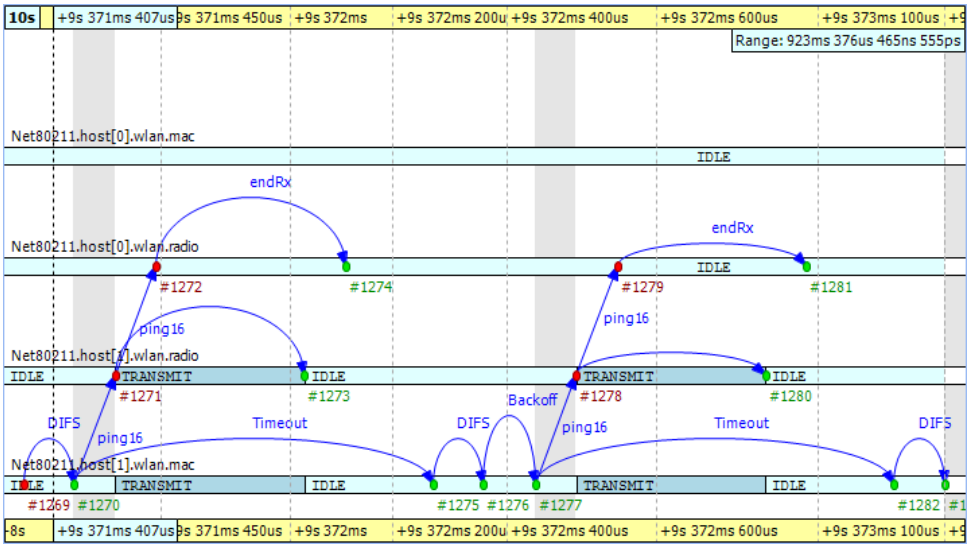


Fig. 8.27: Ping messages get lost

The chart also shows that during the unsuccessful ping period, there are no events occurring in the MAC layer of 'host0,' and it is continuously in the 'IDLE' state.

CHAPTER

NINE

ANALYZING THE RESULTS

9.1 Overview

Analyzing simulation results is crucial for validating models and understanding their behavior. The Analysis Tool in the IDE is an editor that edits an analysis (.anf) file. The analysis file captures the set of inputs and the processing steps necessary for creating the desired plots and tables.

In OMNEST, the results of simulations are captured as scalar values, vector values, statistic summaries (hereafter just "statistics"), and histograms, and are recorded into result files. Result files are tagged with metadata like the network name, iteration variables, and configuration settings. The analysis typically begins with the user specifying a set of result files either directly by name or through patterns that match multiple files. Once loaded, these results can be explored through an intuitive interface where users can browse, filter, transform, and visualize data using various chart types.

Charts can be opened from the selected data with a few clicks. Charts use queries in the form of filter expressions to select the desired subset of results. Users can refine these expressions or write new ones to tailor the data being visualized. The plotting capabilities range from using Matplotlib for complex visualizations to employing the IDE's own Native Plots for more straightforward, interactive displays. Each chart allows adjustments such as labels, colors, line styles, and other visual properties through a configuration dialog.

The charts and their associated settings are remembered as part of the analysis (.anf) file. When the user re-runs the simulations due to modifications in simulation configurations or model adjustments and a new set of result files is created, the analysis tool can automatically fill or refresh the charts with new data.

Each chart in the analysis tool is backed by a customizable Python script and a set of properties that can be edited through a chart configuration dialog. The IDE allows users to edit the chart script to fit their precise needs, to add or modify properties, and even update the configuration dialog to accommodate additional inputs.

Finally, the analysis results can be reproduced and utilized outside the IDE through `opp_charttool`, a command-line tool that recreates chart views from the analysis files. This capability facilitates the integration of results into batch processes, such as compiling LaTeX articles, or sharing them for independent review. Detailed information on this tool and its functionalities is provided in the Simulation Manual, offering a comprehensive framework for managing simulation outputs.

9.2 Creating Analysis Files

The usual way of creating an analysis file is to "imitate" opening an OMNEST result file (`.sca` or `.vec`) by double-clicking it in the *Project Explorer* view. Result files cannot be opened directly, so the IDE will offer creating an analysis file for it instead.

If the result file name looks like the file was created as part of an experiment or parameter study, the IDE creates an analysis file that includes all result files from that experiment as input. In the resulting inputs, the variable part of the file name will be replaced by an asterisk (*), and an input will be added with both the `.sca` and `.vec` file extensions. For example, double-clicking a file called `PureAloha-numHosts=10,iaMean=1-#3.sca` will add `PureAloha-*.sca` and `PureAloha-*.vec` to the analysis.

Upon double-clicking, the *New Analysis File* dialog will open. The folder and the file name are pre-filled according to the location and name of the result file. Press Finish to create the new analysis file.

The same dialog is also available from the menu as *File* → *New* → *Analysis File*. However, analysis files created that way will contain no reference to result files, so file name patterns will need to be added later.

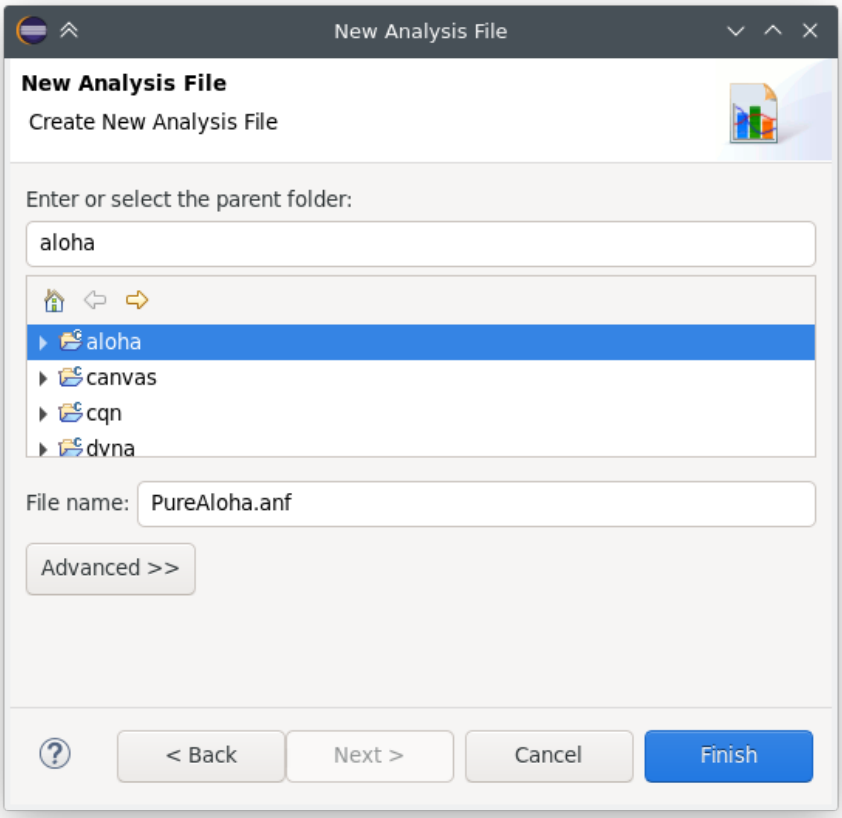


Fig. 9.1: New Analysis File dialog

Tip: If the analysis file already exists, double-clicking on the result file will open it.

9.3 Opening Older Analysis Files

The format of the analysis files (`*.anf`) has changed in OMNEST 6.0 in a non-backward compatible way, meaning that older OMNEST versions will not be able to open new analysis files. OMNEST 6.0, however, attempts to open and convert analysis files created by older versions. Keep in mind that the conversion is a "best-effort" attempt: the result may be incomplete or incorrect. Always check that the converted charts indeed correspond to the original ones, and refine the result if needed.

9.4 Using the Analysis Editor

The usual workflow of result analysis consists of a few distinct steps. These are: adding input files to the analysis, browsing simulation results, and selecting those of interest, creating a chart of an appropriate type from the results, then viewing them as plots, and finally exporting data and/or images if needed.

The Analysis Editor is implemented as a multi-page editor. What the editor edits is the "recipe": what result files to take as inputs, and what kind of charts to create from them. The pages (tabs on the bottom) of the editor roughly correspond to some of the steps described above.

In the next sections, we will go through the individual pages of the editor and which analysis steps can be performed using them.

9.5 The Inputs Page

The first page in the editor is the *Inputs* page, where you specify input files for analysis. You can add a set of file name patterns that specify which result files to load. When the IDE expands the patterns, it displays the list of matched files under each one. The contents of files are also displayed in a tree structure.

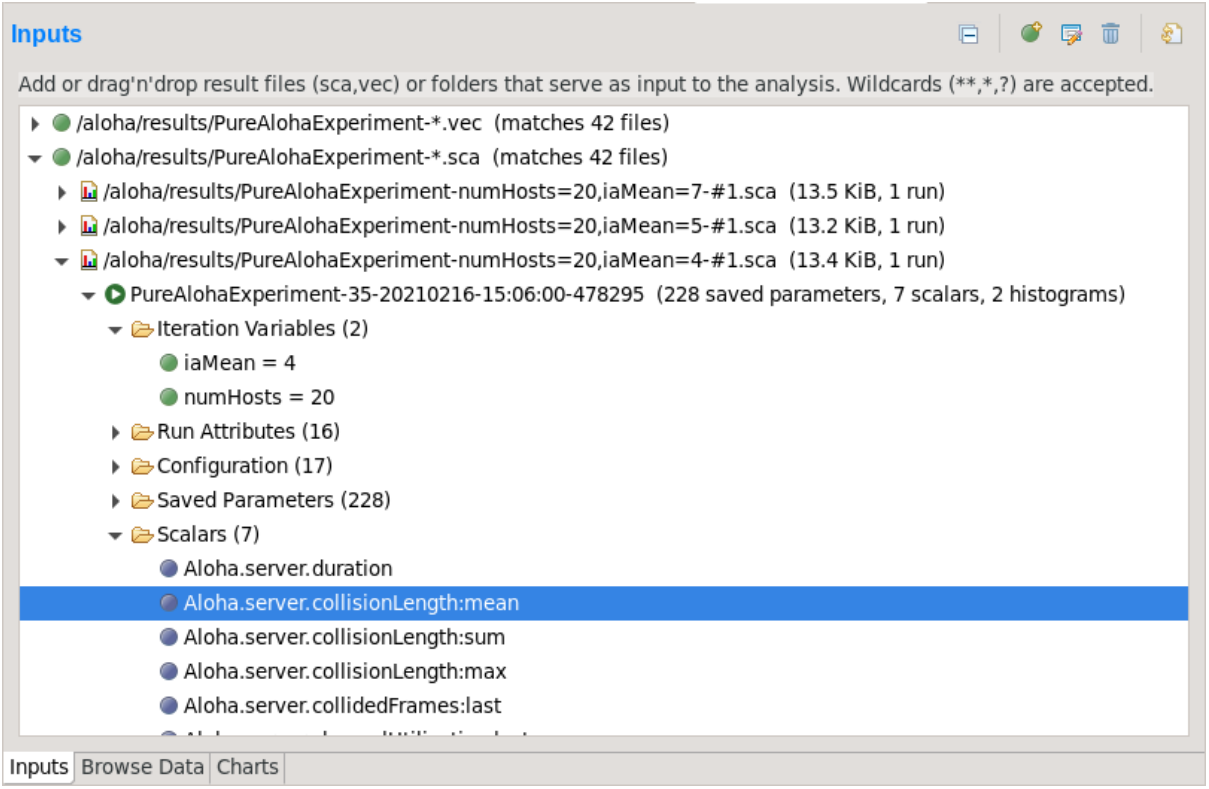


Fig. 9.2: The *Inputs* page

New input files can be added to the analysis by dragging vector and scalar files from the *Project Explorer* view, or by opening a dialog with the *New Input* button on the local toolbar.

9.5.1 Resolution Rules

Input file patterns are resolved with the following rules:

1. An asterisk (`*`) matches files/folders within a single folder.
2. A double asterisk (`**`) may match multiple levels in the folder hierarchy.
3. If the pattern starts with a slash (`/`), it is understood as a workspace full path, with its first component being a project name.
4. If the pattern does not start with a slash (`/`), it is interpreted as relative to the folder of the analysis file.
5. If the pattern identifies a folder, it will match all result files in it (i.e. `/foo/results` is equivalent to `/foo/results/**.sca` plus `/foo/results/**.vec`).

9.5.2 Refresh Files

The input files are loaded when the analysis file is opened.

If files change on the disk or new files are created while the analysis is open (for example, because a simulation was re-run), a refresh can be triggered with the *Refresh Files* button on the toolbar. *Refresh Files* expands the file name patterns again, then loads any new matching files, unloads files that no longer exist on the disk, and reloads the files that have changed since being loaded. Open charts are also refreshed.

Note: In the design of the Analysis Tool, it was a conscious choice to opt for explicit reload in favor of an automatic one. Automatic reload would make it difficult to look at partial results due to excessive refreshing while a large simulation campaign is underway, or when a simulation is continually writing into a loaded vector file.

9.5.3 Reload Files

It is also possible to let the Analysis Tool completely forget all loaded result files, and have them reloaded from scratch. The functionality is available from the context menu as *Reload All Files*.

9.5.4 Are Files Kept in Memory?

The contents of scalar files *are* loaded in memory.

Vector files are not loaded directly; instead, a much smaller index file (`*.vci`) is created and the vector attributes (name, module, run, statistics, etc.) are loaded from the index file. The index files are generated during the simulation, but can be safely deleted without loss of information. If the index file is missing or the vector file was modified, the IDE rebuilds the index in the background.

Tip: The *Progress* view displays the progress of the indexing process if it takes a long time.

9.6 The Browse Data Page

The second page of the Analysis editor displays results (parameters, scalars, histograms, and vectors) from all files in tables and lets the user browse them. Results can be sorted and filtered. Simple filtering is possible with combo boxes, or when that is not enough, the user can write arbitrarily complex filters using a generic pattern-matching expression language. Selected or filtered data can be immediately plotted.

Tip: You can switch between the *All*, *Parameters*, *Scalars*, *Histograms*, and *Vectors* pages using the underlined shortcuts (`Alt+letter` combination) or the `Ctrl+PgUp` and `Ctrl+PgDown` keys.

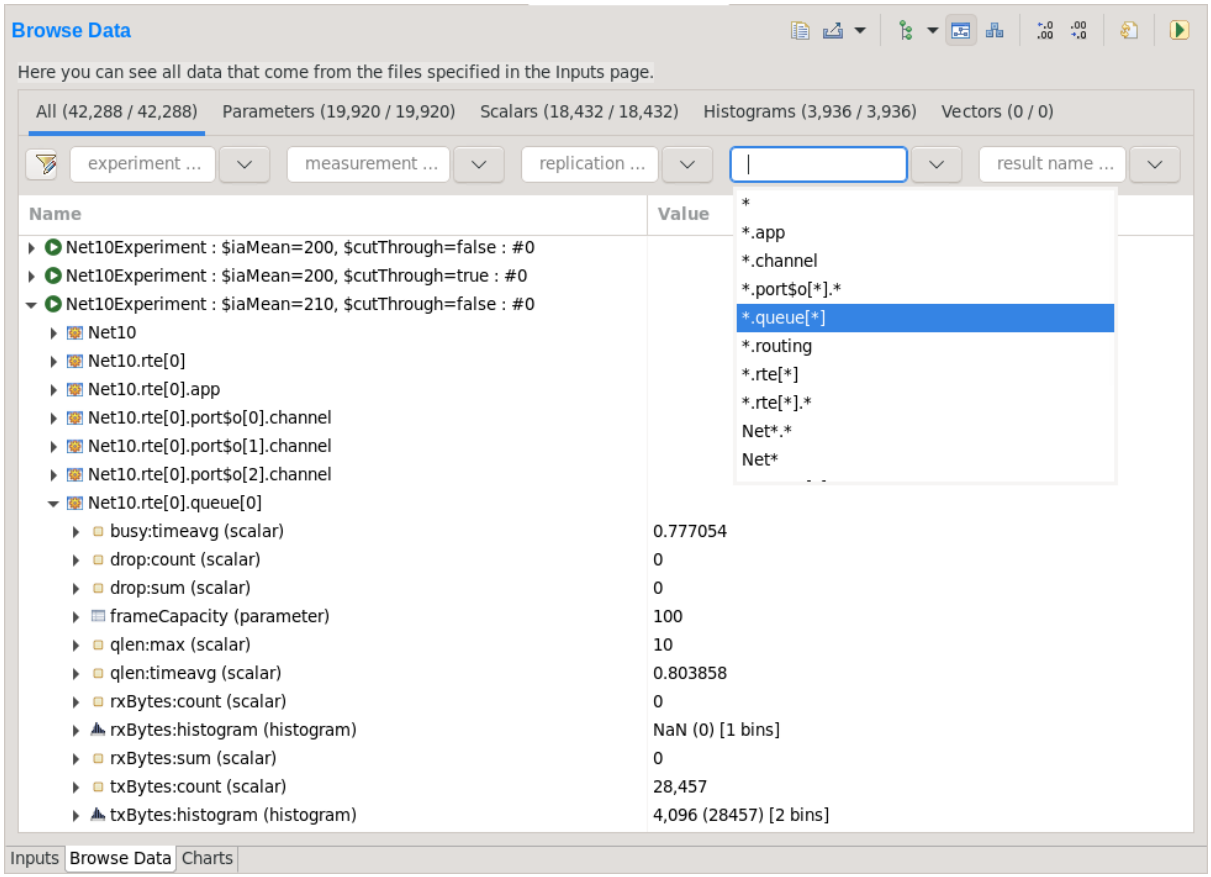


Fig. 9.3: Browsing all data generated by the simulation

The *All* tab shows a tree containing all loaded result items. The structure of this tree can be altered with the *Tree Levels* and *Flat Module Tree* options on the local toolbar and in the context menu.

The other tabs show tables containing the values and attributes of all results of the given type. To hide or show table columns, open *Choose table columns* from the context menu and select the columns to be displayed. The settings are persistent and applied in each subsequently opened editor. The table rows can be sorted by clicking on the column name.

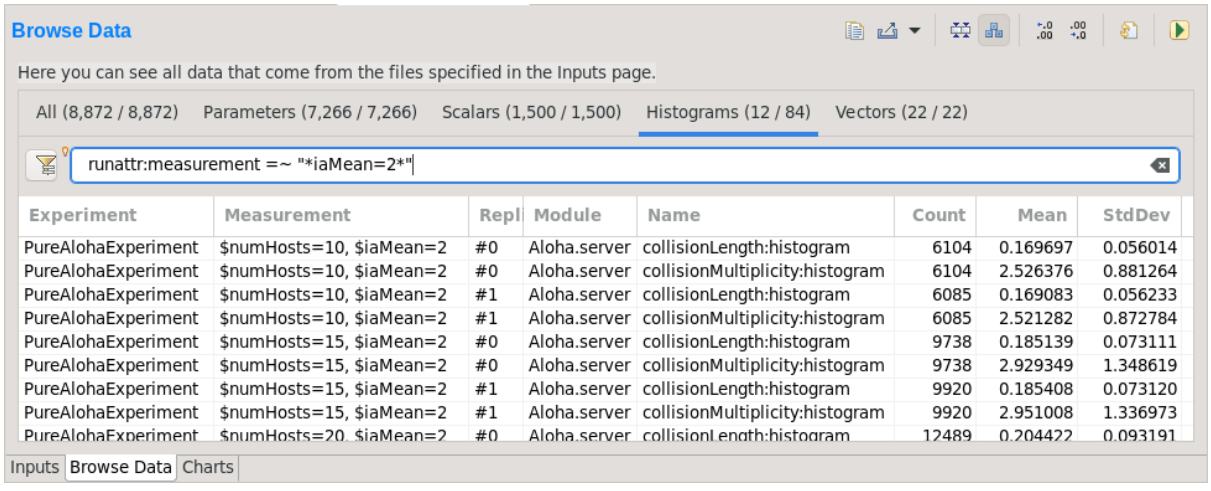


Fig. 9.4: Browsing a subset of result items selected using a filter expression

Individual fields of composite results (e.g. the *:mean* and *:count* fields of statistics, histograms, or vectors) can also be included as scalars by enabling the *Show Statistics/Vector Fields as Scalars* option.

9.6.1 Filtering

Filtering of the table contents is possible with the combo boxes above the tables. The strings in the combo boxes may contain wildcards, and the combo boxes also support content assist (`Ctrl+SPACE`), both of which are useful if there are a huge number of items with different names.

If a more sophisticated selection criteria is needed, it is possible to switch to a more generic filter expression. After pressing the *Filter Expression* button in the filter row, you can enter an arbitrary filter expression. The expression language is described in section *Filter Expressions*.

9.6.2 Plotting

You can display the selected data items on a chart. To open the chart, choose one of the *Plot* items from the context menu, or press Enter (double-click also works for single data lines). See section *Basic Chart Usage* for more information.

9.6.3 Viewing the Details of Result Items

To see the properties of the selected result item, open the *Properties* view. This is useful for checking properties that are not displayed in the table, such as result attributes (`title`, `unit`, `interpolationmode`, etc.), or the full list of bins of a histogram.

9.6.4 Viewing the Contents of a Vector

When selecting a vector, its data can also be displayed in a table. Make sure that the *Output Vector* view is opened. If it is not open, you can open it from the context menu (*Show Output Vector View*). This view always shows the contents of the selected vector.

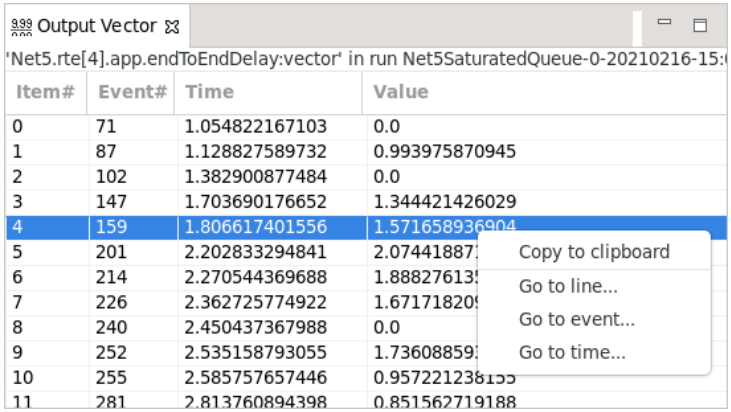


Fig. 9.5: The Output Vector View With its Context Menu

9.6.5 Exporting Data

Selected results can be exported to files in different data formats using the *Export Data* context menu option. After selecting the data format, a dialog to select the output file and configure additional exporting options is shown.

A variety of formats is available, including two CSV-based ones (CSV-R for programmatic consumption and CSV-S for loading into spreadsheets), SQLite, JSON, and so on. Vectors can be also cropped to a time interval in the export.

Tip: You can switch between the *Inputs*, *Browse Data*, and *Charts* pages using the `Alt+PgUp` and `Alt+PgDown` keys.

9.7 The Charts Page

The third page displays the charts created during the analysis.

This page works much like a usual graphical file manager. Each icon represents a chart, and the charts can be selected, reordered by dragging, copied, pasted, renamed, deleted, opened, or their context menu accessed. Different view modes like "icon" and "list" module can be selected.

The *Charts* page also enables you to organize your charts into "folders," providing a more structured and accessible view. This is especially useful when managing a large number of charts.

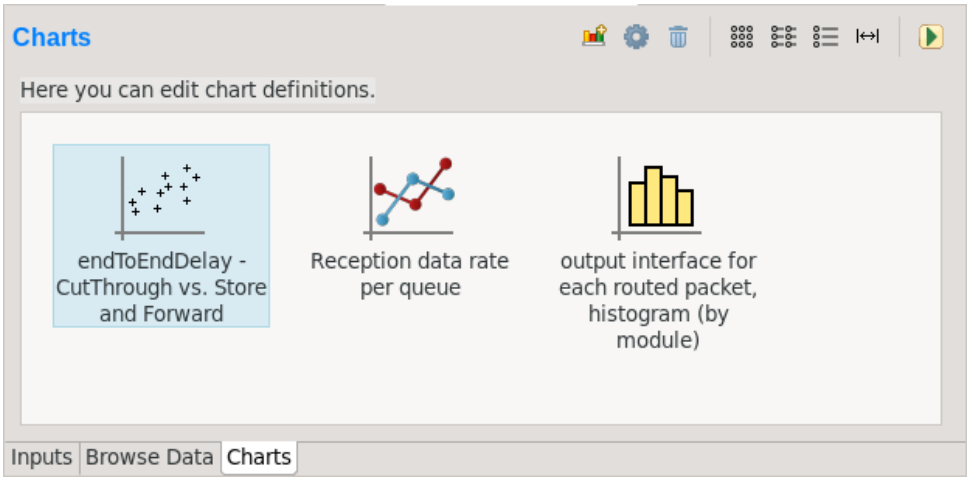


Fig. 9.6: Charts Page

9.8 The Outline View

The *Outline* view shows an overview of the current analysis. Clicking on an element will select the corresponding element in the editor.

Tip: If you select a chart that is currently open, the editor will switch to its page in the editor instead of selecting it in the *Charts* page. If there are many charts open, this can actually be a more convenient way of switching between them than using the tabs at the bottom of the editor window.

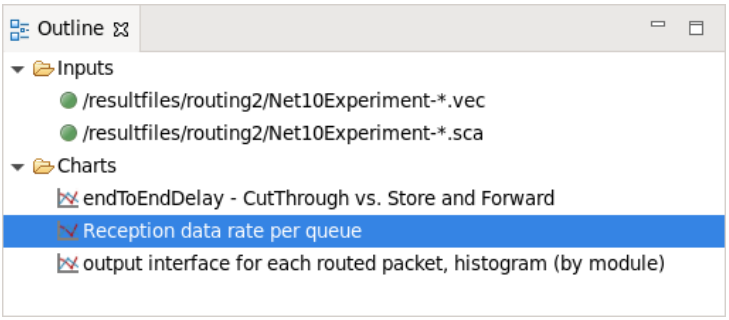


Fig. 9.7: Outline View of the analysis

9.9 Basic Chart Usage

This section introduces you to the basics of working with charts in the OMNEST IDE. It shows how to navigate plots, configure their appearance, and export data and images.

Charts can be created in two ways: first, based on the set of selected results on the *Browse Data* page, and second, choosing from the list of available chart types on the *Charts* page.

9.9.1 Plotting Data

Most often, a new chart is created from a set of simulation results displayed on the *Browse Data* page.

To visualize data, first identify the set of simulation results you wish to plot. By double-clicking on a result item or selecting multiple results and pressing `Enter`, the editor will automatically open an appropriate chart. Alternatively, you can right-click on the selected results to access a context menu that offers a selection of chart templates compatible with the chosen data.

For a more detailed view, select the *Choose from Template Gallery* context menu option. This displays a curated list of templates – filtered to only show those suitable for your data – in the gallery dialog. Here, each template is accompanied by a description and screenshots, providing a comprehensive preview.

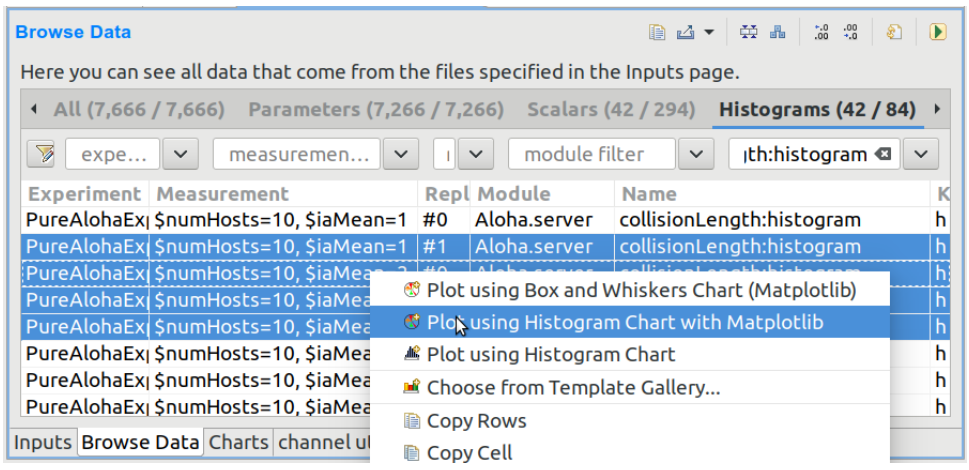


Fig. 9.8: Plotting the selected results

Charts opened this way are *temporary charts*, designed to allow users the flexibility to explore simulation results and their various visualizations without permanent commitment. If you

find a chart that provides valuable insights and wish to keep it for ongoing analysis, you can preserve it by selecting *Save Chart* from the toolbar or the context menu of the chart's page. Once saved, the chart will then be listed on the *Charts* page, making it a permanent part of your analysis.

If you have many charts open, it is easy to lose track of which ones have already been saved into the analysis. To identify if an open chart is temporary, look for the *Save Chart* icon on the leftmost part of the local toolbar. This icon indicates that the chart is temporary. Conversely, if you see the *Go To Chart Definition* icon, the chart has been saved as part of your analysis. Clicking this button will direct you to the *Charts* page, where the saved chart is displayed.

Tip: After saving a temporary chart, it is recommended that you check the filter expression on the *Inputs* page of the chart configuration dialog, and refine or simplify it as needed. When the temporary chart is created, the IDE generates a filter expression based on the selection, but the generated expression is not always optimal, and it may not accurately express your intended selection criteria.

9.9.2 Starting From a Blank Chart

On the *Charts* page, you can create a new chart by right-clicking in an empty area and selecting a chart template from the *New* submenu. Simply clicking on an item from this list will create a new chart based on it.

Alternatively, use the *New Chart* button on the toolbar to open a gallery-like dialog that provides detailed information, including a short description and screenshots, for each chart template. By selecting a template and pressing *OK*, you will instantiate that template into a new chart.

Charts created using either method will initially be empty, as they have not yet been configured with a result selection filter expression.

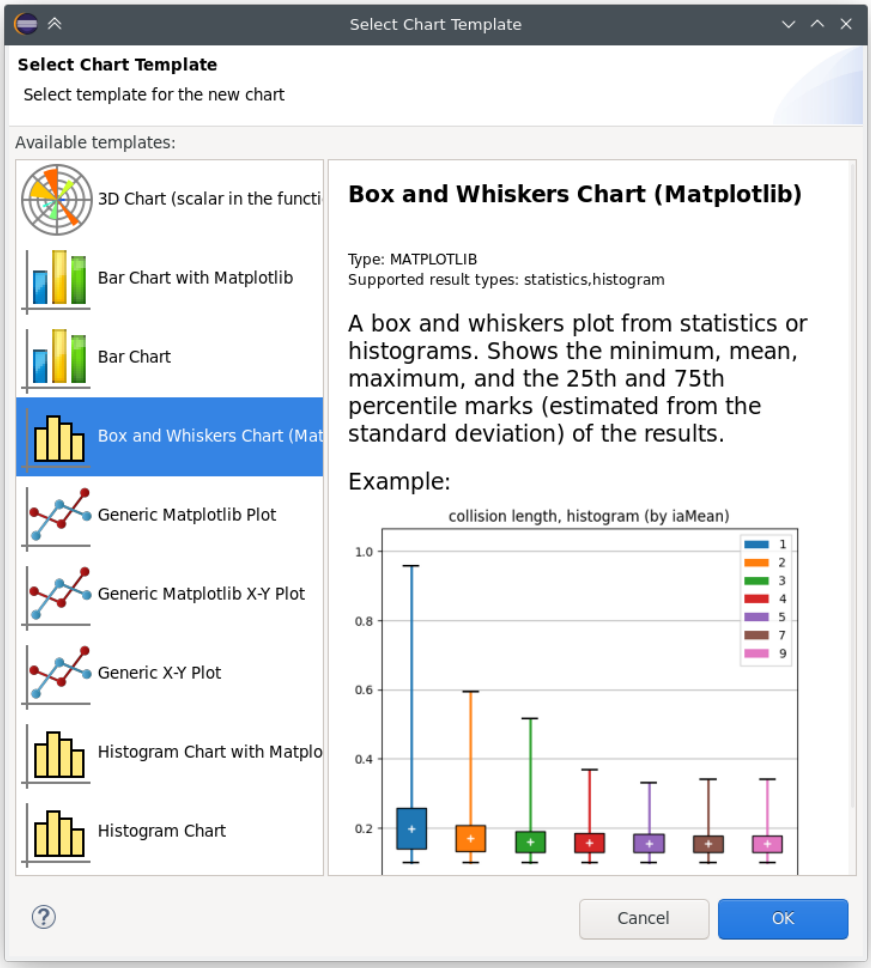


Fig. 9.9: The chart template gallery dialog

9.9.3 Opening an Existing Chart

To open an existing chart, double-click it in the *Charts* page, or select it and hit `Enter`.

9.9.4 Plot Navigation

This section outlines the mouse and keyboard bindings for navigating two types of plots in the Analysis Tool: native plots and Matplotlib-based plots.

- **Native Plots**: These plots support two main modes: Pan mode and Zoom mode. In Pan mode, you can scroll with the mouse wheel and drag the chart. In Zoom mode, the user can zoom in on the chart by left-clicking and zoom out by doing a `Shift` plus left-click or using the mouse wheel. Dragging selects a rectangular area for zooming. The toolbar icons switch between Pan and Zoom modes. You can also find toolbar buttons to zoom in, zoom out, and zoom to fit. Zooming and moving actions are remembered in the navigation history.
- **Matplotlib plots**: Navigation in Matplotlib plots generally follows standard Matplotlib interactions but includes a few enhancements for better control. One addition is the following: the mouse wheel, `Shift` plus mouse wheel, and `Ctrl` plus mouse wheel pans

vertically / horizontally or zooms. There is also a third mode, called Interactive mode, which is used to manipulate interactive elements on the plot, such as widgets, if present.

9.9.5 The Chart Properties Dialog

Charts have a set of properties that define their behavior and appearance. These properties can be edited in a configuration dialog, accessible from the *Configure Chart* toolbar button and context menu item.

The dialog has a tabbed layout, where the list of tabs and the form on each page differ for each chart type. Pages that are common to nearly all chart types (albeit with slightly differing contents) are:

- *Input*: Defines which simulation results should be used as input for the chart and their roles (e.g., which ones to use for the horizontal axis, iso lines, etc.).
- *Plot*, *Lines*, *Bars*, etc.: For configuring the labels, markers, ticks, grid, etc.
- *Styling*: Visual properties for the plot.
- *Advanced*: Lets you manually add plot properties that are not configurable on the other pages.
- *Export*: Properties to be used during image/data export.

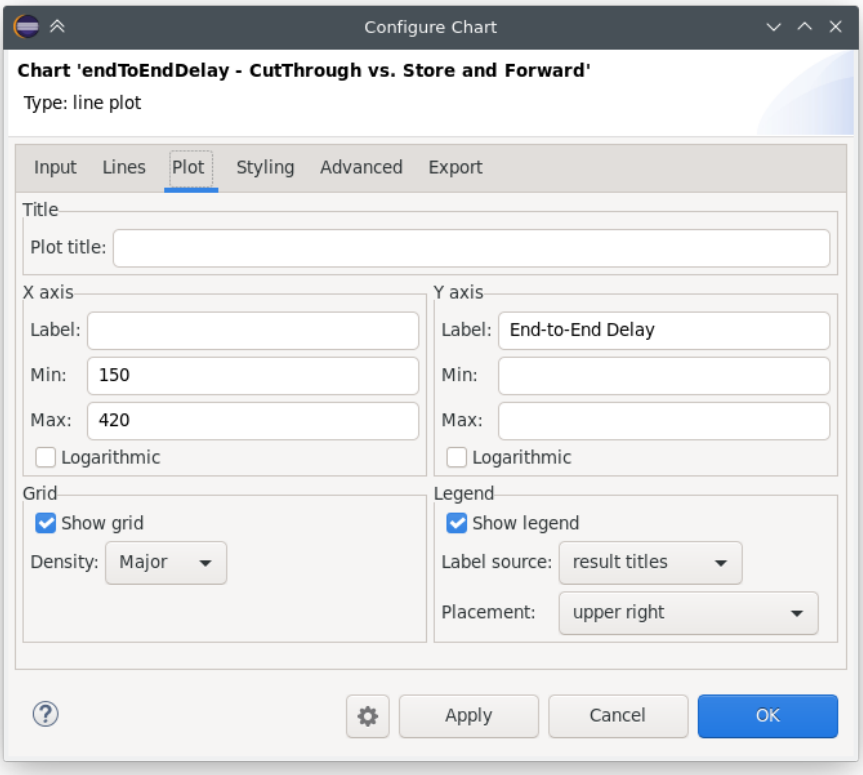


Fig. 9.10: The Chart Properties Dialog

For many input fields, autocompletion and smart suggestions are available by pressing `Ctrl+SPACE`.

9.9.6 Exporting Data

Both the input data used by a chart and the final result after processing can be exported.

The first one is essentially the same as the result exporting option on the *Browse Data* page, except that it uses the result filter expression of the given chart to select which results to export. This is available under the *Export Chart Input As* context menu item of charts.

The second one is available under the common *Export Chart* option, as discussed in section *Batch Export*.

9.9.7 Exporting Images

There are multiple, significantly different ways of exporting a chart to an image:

- You can copy the chart to the clipboard by selecting *Copy to Clipboard* from the context menu. The chart is copied as a bitmap image the same size as the chart on the screen, taking the current navigation state into account.
- The *Save Image* option saves the currently shown part of the chart to an image file. Popular raster and vector formats are accepted, including PNG, JPG, SVG, GIF, TIFF, etc.
- Finally, the *Export Chart* option opens the combined image/data exporting dialog (see section *Batch Export*) for this chart only. This option relies on the chart script for doing the actual exporting.

9.9.8 Batch Export

When exporting multiple charts or when selecting the *Export Chart* option for a single chart, a common export dialog is opened.

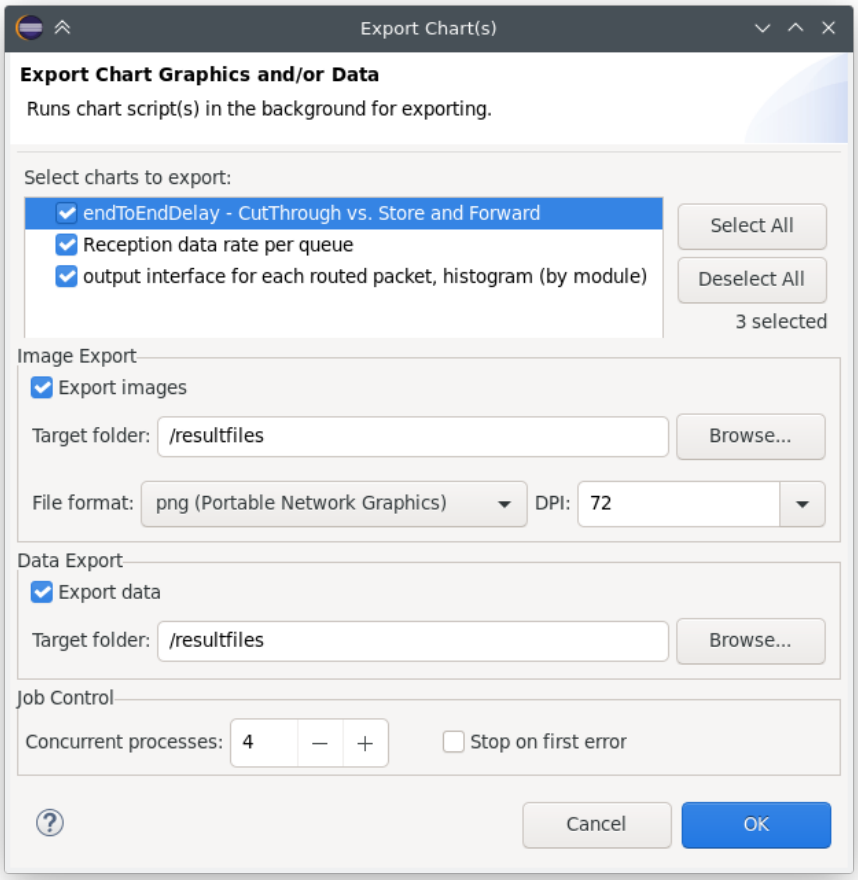


Fig. 9.11: Export Charts Dialog

Individual file names and image dimensions can be specified for each chart in their respective chart configuration dialog.

Note that native charts exported this way will be emulated with Matplotlib, so the saved images may look a bit different than in the IDE.

This is also the way `opp_charttool` exports charts from the command line.

9.10 Configuring Charts

This section discusses working with charts in more detail.

9.10.1 Available Chart Types

The Analysis Tool offers two distinct methods for displaying plots:

- **Matplotlib:** Utilizes the full functionality of Matplotlib within the IDE, allowing for the creation of virtually any type of plot.
- **Native Plots:** Although limited to bar, line, and histogram plots, these widgets are more responsive and scalable compared to Matplotlib.

Chart templates can be categorized according to whether they use native plot widgets for displaying the plot or Matplotlib. It is usually indicated in the name of a chart type whether it is Matplotlib-based or uses a native plot.

There are a number of chart templates in the library of the Analysis Tool. We list the most frequently used ones below.

Based on Native Plots:

- **Line Chart:** Plots vector results as line charts, with the native plot widget. The default interpolation mode is determined automatically from the result attributes. Many kinds of vector operations (smoothing, accumulating, mathematical formulas, etc.) can be easily performed on the vector data. Interpolation, markers, and line style can be configured. Hovering over legend entries with the mouse highlights the corresponding series, clicking on the labels hides/shows the series.
- **Scatter Chart (Scalars in the Function of Itervars):** Displays scalar results on scatter plots, using the native plot widget. X-axis values are taken from a numeric iteration variable. Optionally, results can be grouped into series by iteration variables, run attributes, or result attributes. Markers and line style can be configured. Hovering over legend entries with the mouse highlights the corresponding points/iso line, clicking on the labels hides/shows the points/iso line.
- **Bar Chart:** Plots scalar results as a bar chart, with the native plot widget. The bars can optionally be grouped. Individual bars in each group can be stacked, or positioned in different ways. Hovering over legend entries with the mouse highlights the corresponding data series, clicking on the labels hides/shows the series.
- **Histogram Chart:** Plots histogram results with the native plot widget. The drawing style can be filled or outline. Transformations to cumulative and normalized forms are available. Hovering over legend entries with the mouse highlights the corresponding histogram, clicking on the labels hides/shows the histogram.
- **Histogram from Vectors Chart:** Plots histograms from vector results with the native plot widget. The drawing style can be filled or outline. Transformations to cumulative and normalized forms are available. Hovering over legend entries with the mouse highlights the corresponding histogram, clicking on the labels hides/shows the histogram.

Their Matplotlib equivalents:

- **Line Chart with Matplotlib:** Plots vector results as line charts, with Matplotlib. The default interpolation mode is determined automatically from the result attributes. Many kinds of vector operations (smoothing, accumulating, mathematical formulas, etc.) can be easily performed on the vector data. Interpolation, markers, and line style can be configured.
- **Scatter Chart with Matplotlib (Scalar in the Function of Itervars):** Plots scalar results as scatter charts, using Matplotlib, with the X-axis values taken from a numeric iteration variable. Optionally, results can be grouped into series by iteration variables, run attributes, or result attributes. Markers and line style can be configured. Confidence intervals of averaged points are drawn as error bars.
- **Bar Chart with Matplotlib:** Plots scalar results as a bar chart, with Matplotlib. The bars can optionally be grouped. Individual bars in each group can be stacked, or positioned in different ways. Confidence intervals are displayed as error bars.

- **Histogram Chart with Matplotlib:** Plots histogram results with Matplotlib. The drawing style can be filled or outline. Transformations to cumulative and normalized forms are available.
- **Histogram Chart from Vectors with Matplotlib:** Plots histograms from vector results with Matplotlib. The drawing style can be filled or outline. Transformations to cumulative and normalized forms are available.

Since Matplotlib has vastly more possibilities than the native plots, there are some additional Matplotlib-based charts:

- **Box and Whiskers Chart (Matplotlib):** A box and whiskers plot from statistics or histograms. Shows the minimum, mean, maximum, and the 25th and 75th percentile marks (estimated from the standard deviation) of the results.
- **Line Chart on Separate Axes with Matplotlib:** Plots vector results as line charts, with Matplotlib, each on its own axes. This is very similar to the regular "Line Chart with Matplotlib" template; the only difference is that every vector is drawn into its own separate coordinate system, arranged in a column, all sharing their X axes. The default interpolation mode is determined automatically from the result attributes. Many kinds of vector operations (smoothing, accumulating, mathematical formulas, etc.) can be easily performed on the vector data. Interpolation, markers, and line style can be configured.
- **3D Chart (Scalar in the Function of Itervars):** Plots a scalar result with respect to two iteration variables as a 3D chart. Data points can be rendered as bars, points, or a surface. Various color maps can be chosen.

Generic charts, which can serve as a starting point for custom plots:

- **Generic Matplotlib Plot:** An almost blank template using Matplotlib. It only contains an example script, which you are expected to replace with your own code.
- **Generic Matplotlib X-Y Plot:** An example line plot using Matplotlib. It only contains an example script, which you are expected to replace with your own code.
- **Generic X-Y Plot:** An example line plot using the native plot widget. It only contains an example script, which you are expected to replace with your own code.

The configuration dialog is a little different for each chart type, but they are structured similarly and there are a lot of similarities. The next sections detail how to configure the charts.

Remember that it is straightforward to create new chart templates by customizing existing charts (its chart script and/or the dialog pages) and saving them as a chart template. See the *Editing the Chart Script*, *Editing Dialog Pages*, and *Custom Chart Templates* sections for details.

9.10.2 Defining the Chart Input

Defining the input for the chart is the first step in the process of producing the desired plot. It is normally done on the *Inputs* page of the chart dialog.

The *filter expression* is the most prominent field on the *Inputs* page. It selects from the results loaded into the analysis, that is, from the contents of the result files selected on the *Input* page of the editor. The filter expression can be as simple as `module =~ "*.host[*].app[*]" AND name =~ "pkLatency:mean"` for selecting the mean packet latencies from all apps in the network, or can be composed of many more selectors combined with `AND`, `OR`, and parentheses. The detailed syntax of the filter expression is described in the section *Filter Expressions*.

The filter expression is normally used in a `results.get_vectors()`, `results.get_scalars()`, or `results.get_statistics()` call in the chart script. To see the result of the query in the *Console* view, add the `print(df)` line after the call in the chart script (see "Editing the chart script" section).

In charts working from vector input, the *Inputs* dialog page allows specifying a crop interval and the possibility to leave out empty vectors from the result. These options are implemented as additional arguments to `results.get_vectors()`.

Charts that work from scalar input contain the *Include fields* checkbox, that allows the filter expression to match various fields (min, max, mean, stddev, etc.) of recorded statistics. (Use the *Show fields as scalars* button on the *Browse Data* page to see them.) This is also implemented as an additional argument to `results.get_scalars()`.

To include additional input, modify the Python script to add your own data. Use cases: To use multiple filter expressions (and combining the results); to add external reference data; to compute new scalars from vectors or other scalars as input.

After executing a result query, most charts require additional processing before the data can be visualized. Charts utilizing scalar data typically involve a *pivoting* step, while those working with vector data may incorporate *vector operations* such as summation, computing running averages, or window averages. Details on pivoting and vector operations will be covered in subsequent sections. However, we will first explore the syntax of the filter expression in detail.

9.10.3 Filter Expressions

Filter expressions are primarily used on the *Input* page of chart dialogs for selecting simulation results as input for the chart. They can also be used on the *Browse Data* editor page for filtering the table/tree contents, and they also appear, in more generic forms, in other parts of the IDE.

A filter expression is composed of terms that can be combined with the `AND`, `OR`, `NOT` operators, and parentheses. A term filters for the value of some property of the item and has the form `<property> =~ <pattern>`, or simply `<pattern>`. The latter is equivalent to `name =~ <pattern>`.

A typical example is to select certain simulation results recorded by specific modules. For example, the expression `module =~ "**.app[*]" AND name =~ "pkRecvd*"` selects results whose name begins with `pkRecvd` from modules whose name is `app[0]`, `app[1]`, etc.

Patterns only need to be surrounded with quotes if they contain whitespace or other characters that would cause a parsing ambiguity.

Here is the full list of available properties:

- `name`: Name of the result or item.
- `module`: Full path of the result's module.
- `type`: Type of the item. The value is one of: `scalar`, `vector`, `parameter`, `histogram`, `statistics`.
- `isfield`: `true` if the item is a synthetic scalar that represents a field of a statistic or a vector, `false` if not.
- `file`: File name of the result or item.
- `run`: Unique run ID of the run that contains the result or item.
- `runattr:<name>`: Run attribute of the run that contains the result or item. Example: `runattr:measurement`.
- `attr:<name>`: Attribute of the result. Example: `attr:unit`.
- `itervar:<name>`: Iteration variable of the run that contains the result or item. Example: `itervar:numHosts`.
- `config:<key>`: Configuration key of the run that contains the result or item. Example: `config:sim-time-limit`, `config:**.sendIaTime`.

In the values, the match pattern may contain the following wildcards:

- `?` matches any character except '.'
- `*` matches zero or more characters except '.'
- `**` matches zero or more characters (any character)
- `{a-z}` matches a character in range a-z
- `{^a-z}` matches a character not in the range a-z
- `{32..255}` any number (i.e., sequence of digits) in the range 32..255 (e.g., `99`)
- `[32..255]` any number in square brackets in the range 32..255 (e.g., `[99]`)
- `\\` takes away the special meaning of the subsequent character

---

Tip: Content Assist is available in text fields where you can enter filter expressions. Press `Ctrl+SPACE` to get a list of appropriate suggestions at the cursor position.

9.10.4 Pivoting

Charts utilizing scalar data, such as bar charts and scatter plots, typically involve a *pivoting* step, which converts the data from a linear, list-like format into a more structured table format, which is essential for these types of visualizations.

The `results.get_scalars()` call produces a data frame with the essential columns `module`, `name` (result name), and `value`, along with additional columns for potential result attributes and various properties describing the simulation run. After pivoting along the `module` and `name` columns, this data is transformed so that each module becomes a row, each result name becomes a column, and the values fill the cells at the intersection of these rows and columns. If the data includes results from multiple simulations, the values are averaged to provide a consolidated overview.

9.10.5 Vector Operations

The charts that show vector results offer a selection of operations to transform the data before plotting.

These can be added to the chart under the *Apply* or *Compute* context menu items. Both ways of adding operations compute new vectors from existing ones. The difference between them is that *Apply* replaces the original data with the computation result, while *Compute* keeps both.

Some operations have parameters that can be edited before adding them.

The operations are added to a field on the *Input* page of the chart configuration dialog.

Most operations perform a fairly simple transformation on each individual vector independently.

For example, see the screenshots illustrating the effects of the following vector operations:

```
apply:sum
apply:diffquot
apply:movingavg(alpha=0.05)
```

The operations `apply:sum`, `apply:diffquot`, and `apply:movingavg(alpha=0.05)` transform vector data by computing cumulative sums, rate of change between consecutive values, and applying an exponentially weighted moving average, respectively.

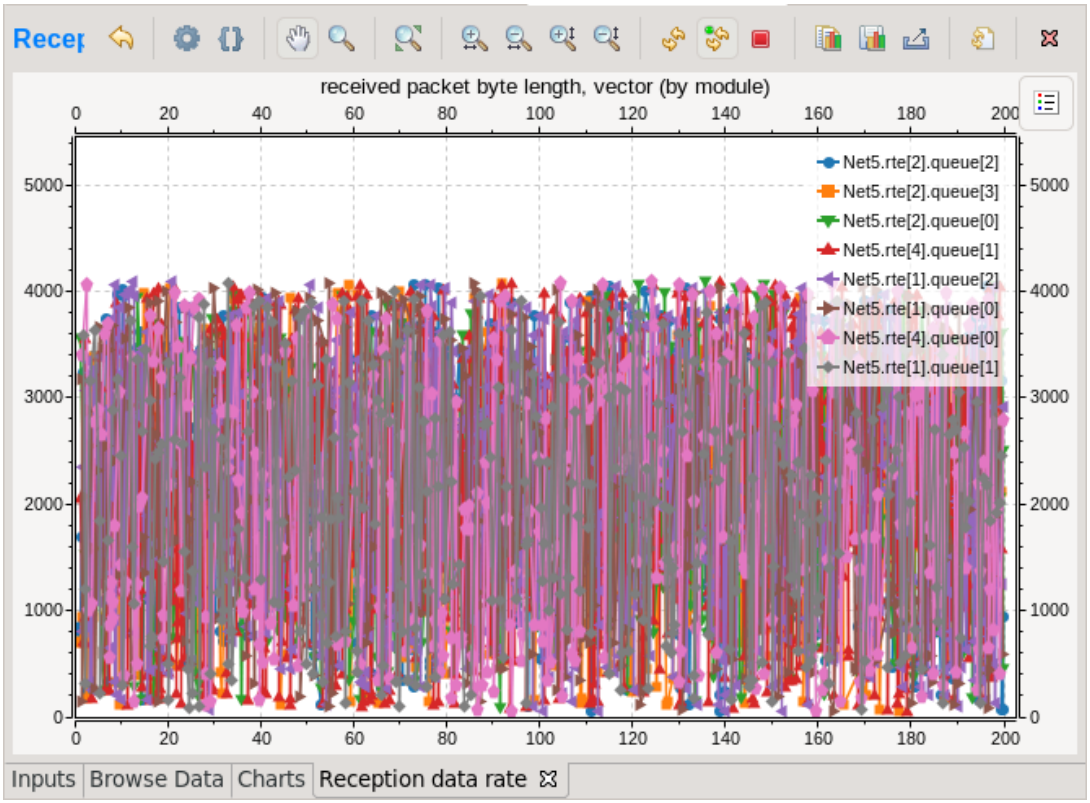


Fig. 9.12: Vector Operations - Before

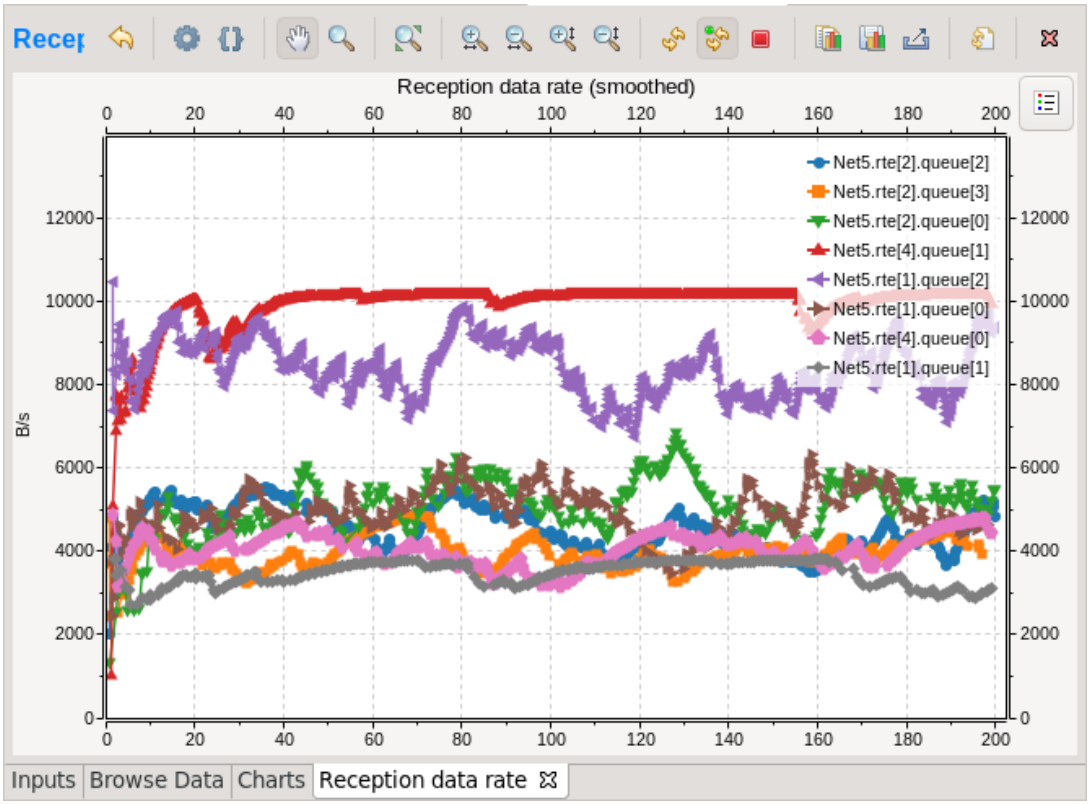


Fig. 9.13: Vector Operations - After

The list of available operations includes:

- *mean()*: Computes the cumulative average of values up to each point.
- *sum()*: Calculates the cumulative sum of values up to each point.
- *add(c)*: Adds a specified constant to all values.
- *compare(threshold, less=None, equal=None, greater=None)*: Compares each value against a threshold and replaces it based on specified conditions.
- *crop(t1, t2)*: Discards values outside a specified time interval.
- *difference()*: Subtracts each value from its predecessor.
- *diffquot()*: Computes the rate of change between consecutive values.
- *divide_by(a)*: Divides all values by a constant.
- *divtime()*: Divides each value by its corresponding time.
- *expression(expression, as_time=False)*: Evaluates a Python expression for each value, optionally updating time instead of values.
- *integrate(interpolation="sample-hold")*: Integrates the series using the specified interpolation.
- *lineartrend(a)*: Adds a linear trend to the series.
- *modulo(m)*: Applies modulo operation to the series with a constant.
- *movingavg(alpha)*: Applies an exponentially weighted moving average to the series.
- *multiply_by(a)*: Multiplies all values by a constant.
- *removerepeats()*: Removes consecutive repeated values.
- *slidingwinavg(window_size, min_samples=None)*: Computes the average of values within a sliding window.
- *subtractfirstval()*: Subtracts the first value from all subsequent values.
- *timeavg(interpolation)*: Computes the average of values over time using the specified interpolation.
- *timediff()*: Calculates the time difference between consecutive values.
- *timeshift(dt)*: Shifts the time series by a constant.
- *timedilation(c)*: Scales the time series by a constant factor.
- *timetoserial()*: Converts time values to their sequential index.
- *timewinavg(window_size=1)*: Computes the average of values within a fixed time window.
- *timewinthruput(window_size=1)*: Calculates the throughput over a fixed time window.
- *winavg(window_size=10)*: Computes the average of values within each batch of a specified size.

See a description of all built-in vector operations in the Simulation Manual.

9.10.6 Plot Options

The configuration dialogs for charts contain specific pages tailored to customizing the plot:

- The *Plot* page allows setting the plot title, adjusting axis labels, setting axis limits, configuring axis scales (linear or logarithmic), toggling and configuring grid display, and managing legend display and placement.
- The *Lines* page appears in line plots, and allows you to customize plot line attributes such as style, color, and width, and marker characteristics including type and size.
- The *Bars* page appears with bar charts, and allows changing the baseline, the bar placement (aligned, overlap, in-front, or stacked), and details like label rotation.
- The *Histogram* page appears in histogram plots, and allows the user to configure histograms by setting a baseline, choosing between solid or outline draw styles, normalizing data, displaying cumulative results, and managing under/overflows.

The majority of the settings mentioned are straightforward and intuitive; however, there is an important aspect regarding how colors and markers are determined in the plots. Unlike static configurations, the number of data items represented in the plot is dynamic, varying based on the results retrieved by a query. Consequently, colors and markers cannot be assigned directly to each individual data point.

Instead, these visual attributes are managed through "cyclers," which systematically rotate through a predefined set of colors and markers. This approach ensures an appealing visual representation regardless of the number of data items displayed. To customize the sequence of colors and markers used in Auto mode, you can adjust the cycle seed on the *Styling* page. This allows for the modification of the appearance of plot elements dynamically, accommodating the varying result sets returned by different queries.

9.10.7 Legend Labels

The labels of data items in the legend are normally produced automatically, making use of the properties that differ across the data items. (The properties that are the same in all items are, on the other hand, used for producing the chart title.) With automatic legend labels, the user is given the choice of stating the preference between using result names instead of result titles, and module display paths instead of module full paths. (The result title is the content of the `title` attribute of the result. The display path is a variant of the full path where, if available, the display names of modules are used instead of the normal names; the display name is set using the `display-name` configuration option.)

For those who require more detailed control, the Manual mode allows users to define a custom format string for the legend labels. This string can include placeholders like `$name`, `$title`, `$module`. These placeholders refer to dataframe columns, so the exact list varies depending on the chart type and the kind of simulation results. When in doubt, insert a `print(df)` statement in the chart script and check the log in the Console.

The labels produced like that can be further tweaked using replacements. You can input plain substrings or regular expressions to be replaced with the strings you specify. Using this feature, you can achieve things like replacing abbreviations with full terms, discarding unwanted parts, replacing module names with more descriptive names, or adjusting separator/punctuation characters or spacing. For example, the `/host\[(\d+)\]/Host \1/` regex replacement will turn strings like `host[0]`, `host[1]`, etc. into `Host 0`, `Host 1`, and so on.

9.10.8 Ordering

Charts normally allow controlling the order of the data items (series) in the plot. The ordering affects both the chart presentation and the legend, enabling users to place important or related items together.

When exporting multiple charts, or when selecting the *Export Chart* option for a single chart, a common export dialog is opened.

The order is defined via a list of regular expressions that are matched against the legend labels of the items. The plot items will be ranked based on the index of the regular expression the item first matches. Case-sensitive substring match is done, so `^` and `$` should be used to match the beginning and end of the label, respectively. For example, the regex list (`router`, `host`) will place all items whose label contains the "router" string in front of items that contain "host", and items that contain neither will follow. The list (`^B`, `^A`) will move items starting with capital "B" to the top, followed by items starting with capital "A", and the rest below.

There are two regular expression lists, defining a primary and secondary ordering. The primary ordering takes precedence, and the secondary ordering is used to further refine the arrangement of items that are equivalently ranked in the primary order.

A further checkbox allows users to enable or disable alphabetical sorting as a tertiary ordering mechanism. This is useful when two items do not match any of the specified regular expressions, ensuring that there is still a consistent rule to fall back on for their ordering. When activated, this setting ensures that after considering the regex-based rankings, items will be alphabetically ordered.

9.10.9 Styling

The *Styling* page of the dialog allows setting a number of options that affect the presentation of the plot.

For Matplotlib-based charts, you can select the plot style. This is the same that you can select in plain Matplotlib using the `matplotlib.style.use(style)` command. There are a number of built-in styles, and you can add new styles by installing packages like `seaborn` or `prettyplotlib`.

You can set the background colors, some legend display options, etc.

You can set the seed used for the color and marker cyclers. Experimenting with different seeds allows you to choose a new set of colors/markers for the plot if you do not like the default ones. If you want to have even more control over the colors and markers, you can define your own cycler and enter it as properties on the *Advanced* page of the dialog.

The *Advanced* page enables even more fine-grained customization by allowing users to directly set visual plot properties that are not explicitly configurable in the dialog. For Matplotlib charts, you can enter settings in the format known as "rcParams" in Matplotlib terminology. Native plots have their own visual properties; content assistance in the dialog will help discover them. Native plots also allow directly setting colors for individual items via properties.

9.11 Editing the Chart Script

All charts are powered by Python scripts, which take their configuration settings from properties that can be edited in the *Chart Configuration* dialog. All of these elements are under your full control so that you can create exactly the plots that you need for your analysis: you can edit the chart script, you can edit the properties using the configuration dialog, and you can also modify/tweak the configuration dialog itself to add input fields for extra properties, for example. Each chart has its own copy of everything (the chart script, properties and config dialog pages), so modifying one chart will not affect other similar charts.

9.11.1 Editing

To see or edit the chart's Python script, click the *Show Code Editor* button on the toolbar of an open chart. With the code editor open, you are free to make any changes to the chart's script.

The integrated editor is that of the PyDev project. It provides syntax highlighting, code navigation (go to definition, etc.), helpful tooltips (using docstrings), and content assist (completion suggestions).

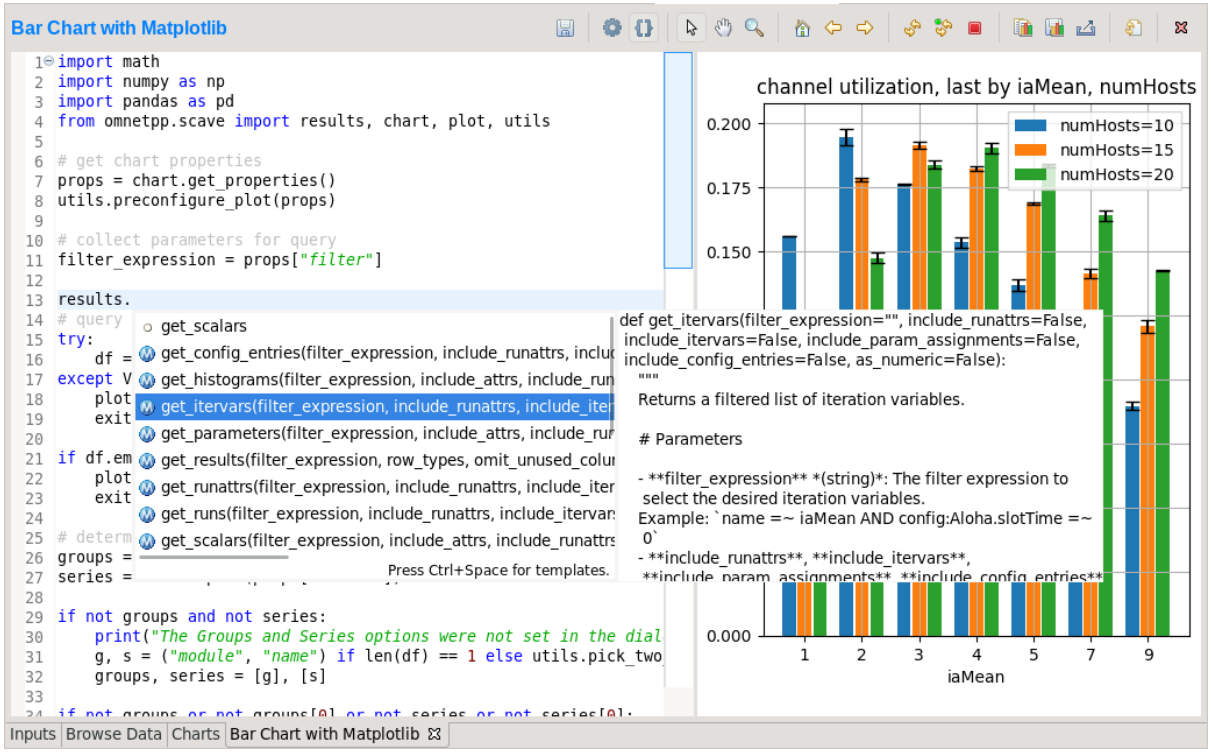


Fig. 9.14: Chart Script Editor

9.11.2 Refreshing the Chart

Normally, the chart script is automatically re-executed with some delay after each edit. This functionality can be enabled/disabled using the *Automatic Refresh* button on the chart page toolbar. Independent of the auto-refresh state, you can always trigger a manual refresh (re-execution of chart script) by pressing the *Refresh* on the toolbar. If the chart script execution takes too long, you can abort it by clicking the *Kill Python Process of the Chart* button on the toolbar.

Tip: The viewport (zoom/pan state) is usually preserved after refresh. If the area occupied by the displayed data changes significantly for some reason, it is possible that you will see an empty plot after the refresh, simply because valuable content now falls outside the viewport. Push the *Home* icon on the toolbar in these cases to bring all plotted elements into view.

9.11.3 Console Output

The console output of the script, i.e. text written to the *stdout* and *stderr* streams, is displayed in the *Console* view. Each chart has a console of its own in the view, which is activated when switching to the chart's page in the editor. Text written to the standard error stream appears in red. You can write to the console using Python's `print()` statement. Notably, `print(df)` is a very useful line that you'll probably end up using quite often.

Note: Even though PyDev offers a variety of tools for debugging Python scripts, these unfortunately don't work on chart scripts. Limited debugging can be performed using print statements, throwing exceptions, and dumping stack traces, which is usually enough. If you really need debugging to get a piece of code working, one way is to factor out the code to be able to run independently, and use an external debugger (or the IDE's debugger) on the resulting `.py` file.

9.11.4 Errors

Errors are marked in the source code with a red squiggle and a sidebar icon. Hover over them to see a tooltip describing the error. The errors are also entered into the *Problems* view. Double-clicking these problem entries will reveal the line in the code editor where the error came from. Errors marked this way include Python syntax errors, and runtime errors that manifest themselves in the form of Python exceptions. For exceptions, the stack trace is printed in the *Console* view.

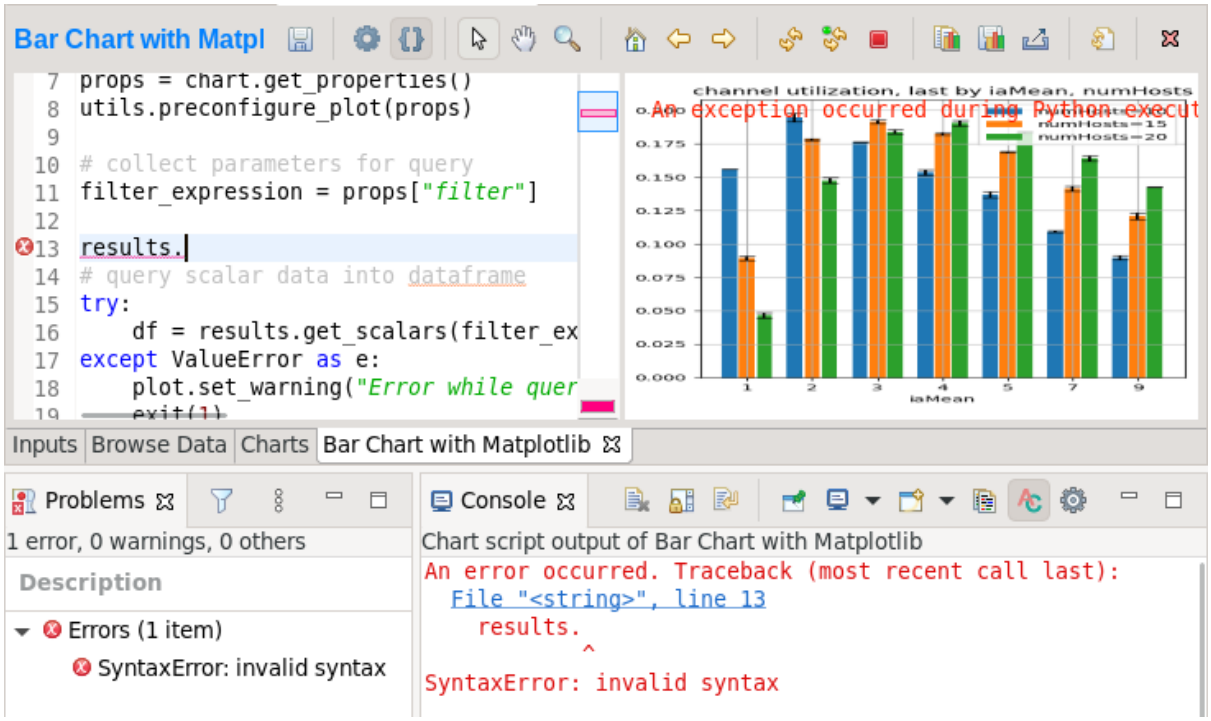


Fig. 9.15: A Python error is marked on the GUI

9.12 Editing Dialog Pages

9.12.1 The Edit Pages Dialog

If you need to add support for new configuration properties to the chart, you will need to edit the forms on the *Configure Chart* dialog. Pages (tabs) in the configuration dialog are represented as XSWT forms. To see or edit the pages and forms within, click the *Edit Dialog Pages* button on the property editor dialog.

The action will bring up the *Edit Chart Dialog Pages* dialog, which lets you edit the forms that make up the configuration dialog of the chart. You can add, remove, reorder, and rename tabs, and you can edit the XSWT form on each tab. A preview of the edited form is also shown.

9.12.2 XSWT Page Descriptions

XSWT is an XML-based UI description language for SWT, the widget toolkit of Eclipse on which the OMNEST IDE is based. The content of XSWT files closely mirrors SWT widget trees.

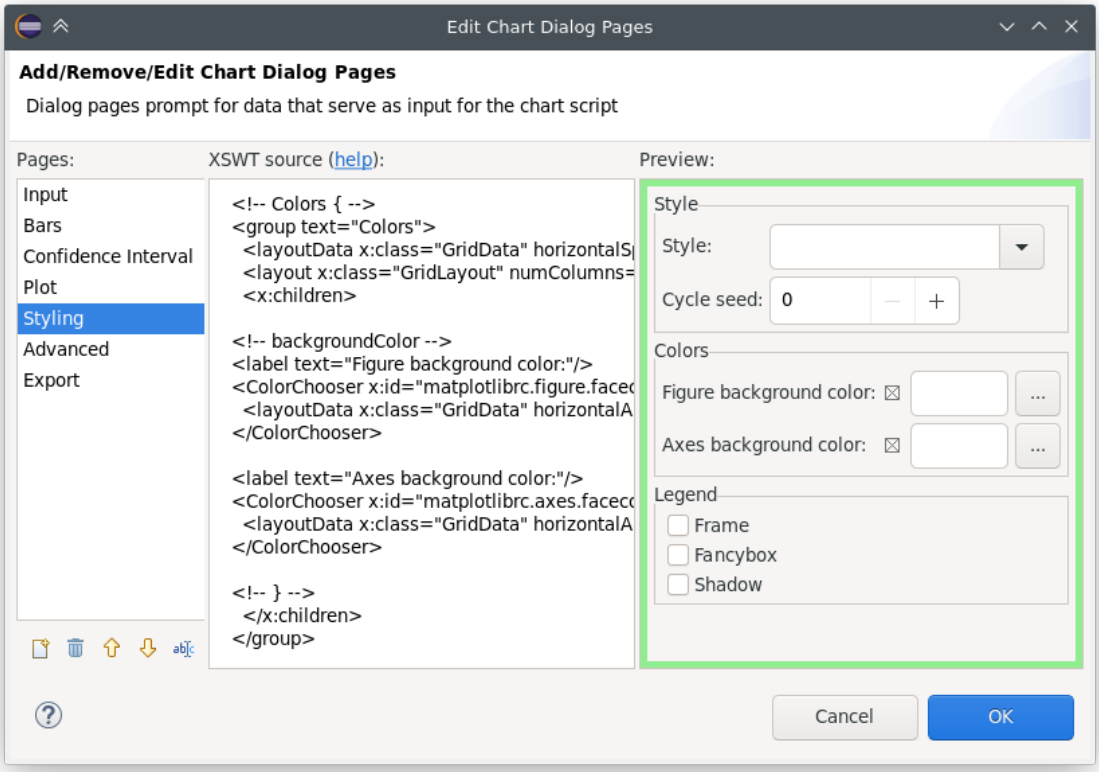


Fig. 9.16: Editing Chart Properties Editor Pages

The *New Page* in the dialog brings up a mini wizard, which can create a full-fledged XSWT page from a shorthand notation of its content provided by you.

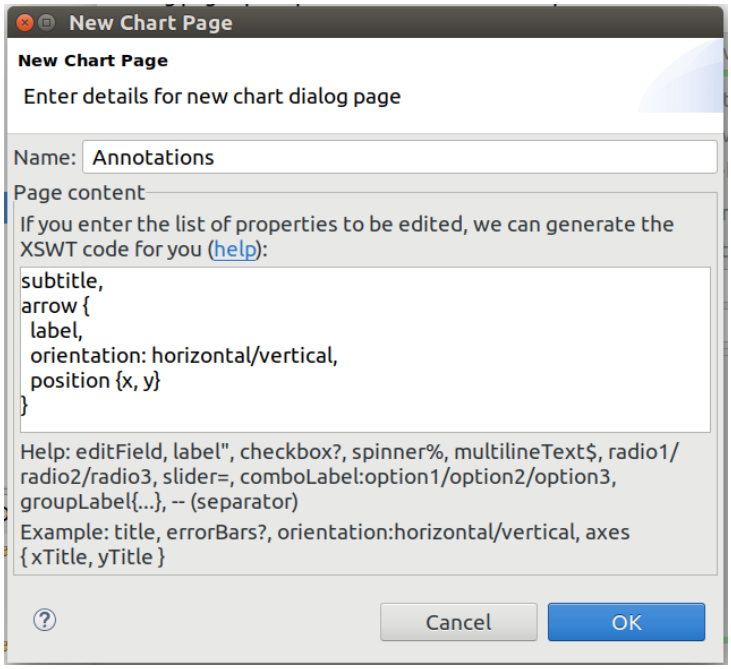


Fig. 9.17: The Creating a New Dialog Page From a Shorthand Notation

Some XML attributes in the XSWT source have special roles:

- `x:id` binds the contents of the widget to a chart property. For example, an edit control defined as `<text x:id="title">` edits the `title` chart property, which can be accessed as `props["title"]` in the chart script.
- `x:id.default` provides a default value for the chart property named in the `x:id` attribute.
- Further `x:id.*` attributes are also used, e.g., `x:id.contentAssist` defines the kind of content assist requested for the edit control, or `x:id.isEnabler` denotes a checkbox as the enabler of the widget group that contains it.

Tip: The easiest way to add a new field to a page is to look at other pages (or other charts' pages) and copy/paste from them.

9.13 Chart Programming

Data processing in chart scripts is based on the NumPy and Pandas packages, with some modules provided by OMNEST.

9.13.1 Python Modules

The chart scripts can access some functionality of the IDE through a couple of modules under the `omnetpp.scave` package. These include: `chart`, `results`, `ideplot`, `vectorops`, and `utils`. The complete API of these modules is described in the Simulation Manual.

The `chart` module exposes information about the chart object (as part of the analysis, and visible on the *Charts* page), most importantly its set of properties but also its name and what type of chart it is.

The `results` module provides access to the set of result items (and corresponding metadata) currently loaded in the analysis in the IDE. This data is accessible through a set of query functions, each taking a filter expression, and returning a Pandas DataFrame.

The `ideplot` module is the interface for displaying plots using the IDE's native (non-Matplotlib) plotting widgets from chart scripts. The API is intentionally very close to `matplotlib.pyplot`. When `ideplot` is used outside the context of a native plotting widget (such as during the run of `opp_charttool`, or in the IDE during image export), the functions are emulated with Matplotlib.

The `vectorops` module contains the implementations of the built-in vector operations.

The `utils` module is a collection of utility functions for processing and plotting data. Most chart scripts heavily rely on `utils`.

Additionally, the well-known `numpy`, `pandas`, `matplotlib`, and sometimes the `scipy` and `seaborn` packages are often utilized. All other packages installed on the system are also fully available.

Tip: See the Simulation Manual for details on the OMNEST result analysis Python modules. It contains a section on chart programming, and an API reference in the Appendix.

9.13.2 Tips and Tricks

This section is a collection of tips for use cases that might come up often when working with charts, especially when editing their scripts.

Sharing Code Among Charts

For future releases, we are planning to support "snippets" as part of the analysis file, as a means of sharing code among charts. Until that feature is implemented, a workaround is to put shared code in `.py` files. These scripts can be imported as modules. They will be looked for in the folder containing the `.anf` file and in the `python` folders of the containing project and all of its referenced projects. Chart scripts can import these files as modules and thereby use the functionality they provide. This also makes it possible to use external code editors for parts of your code.

Adding Extra Data Items to the Plot

It's possible to add new data items to the queried results before plotting. These can be computed from existing items or synthesized from a formula. Example uses:

- Computing derived results:

  ```
  df["bitrate"] = df["txBytes"] / df["sim-time-limit"]
  ```

- Adding analytical references, like theoretical values in an ideal scenario:

  ```
  df["analytical"] = df["p"] * (1 - df["p"]) ** (df["N"]-1)
  ```

- Summarizing results:

  ```
  df["mean"] = df["vecvalues"].map(np.mean)
  ```

Simplifying Complex Queries

Instead of coming up with an elaborate filter expression, it is sometimes more straightforward to query results multiple times within a script and combine them with `pd.concat`, `pd.join`, or `pd.merge`. Other functions like `pf.pivot` and `pd.pivot_table` are also often useful in these cases.

Defining New Vector Operations

You can define your own vector operations by injecting them into the `vectorops` module, even if this injection is done in an external module (`.py` file imported from the directory of the `.anf` file).

```
from omnetpp.scave import vectorops
def myoperation(row, sigma):
   row["vecvalue"] = row["vecvalue"] + sigma
   return row
vectorops.myoperation = myoperation
```

After injection, use it like any other vector operation, on the *Input* page of Line Charts for example: `apply: myoperation(sigma=4)`

Customized Export

If the built-in image/data exporting facilities are not sufficient for your use case, you can always add your export code, either by manually `open()`-ing a file or by utilizing a data exporter library/function of your liking. Functions such as `plt.savefig()` and `df.to_*()` can be useful for this.

Caching the Result of Expensive Operations

Since the entire chart script is executed on every chart refresh, even if only a visual property has changed, it can sometimes help to cache the result of some expensive data querying or processing procedure in the script. And because every execution is in a fresh Python process, caching can only really be done on the disk.

There are existing packages that can help you with this, such as `diskcache`, `cache.py`, or `memozo`. (Note that caching the result of a function call is often called *memoization*; using that term in online searches may give you additional insight.)

If the sequence of operations whose result is cached includes simulation result querying (`results.get_scalars()`, etc.), it is important to invalidate (clear) the cache whenever there

is a change in the loaded result files. The change can be detected by calling the `results.get_serial()` function, which returns an integer that is incremented every time a result file is loaded, unloaded, or reloaded.

Arbitrary Plot Types

In charts using Matplotlib, the whole range of its functionality is available:

- Arbitrary plots can be drawn (heatmaps, violin plots, geographical maps, 3D curves, etc.)
- Advanced functionality like mouse event handlers, graphical effects, animations, and widgets all work
- It's also possible to just add small customizations, like annotations
- Any extension library on top of Matplotlib can be used, such as: *seaborn*, *ggplot*, *holoviews*, *plotnine*, *cartopy*, *geoplot*
- The built-in plotting capability of Pandas DataFrames (under `df.plot`) works too

Per-Item Styling on Native Plots

For native plots, properties affecting individual data items can be specified with the following additional syntax: `<propertyname>/<itemkey>`. Unless overridden manually, the data item keys are sequentially increasing integers, starting with `1`. For example, adding the following line on the Advanced tab in the property editor dialog of a line chart will set the color of the second line (or of the line identified with the key `2`) to red.

```
Line.Color/2 :  #FF0000
```

9.14 Custom Chart Templates

When charts are created, they are instantiated from a template. The list of available chart templates can be browsed in the template gallery dialog, available from the *Charts* page as *New Chart* and from the *Browse Data* page as *Choose from Template Gallery*. The dialog shows some properties (chart type, accepted result types), a description, and often also sample images for each one.

The IDE contains a number of built-in chart templates, but the user can add their own too. Custom chart templates live in the `charttemplates` folder of every project and are available in analyses in the same project and all projects that depend on it.

9.14.1 Exporting a Chart as Template

The easiest way of creating a custom chart template is by customizing a chart, then saving it as a template. The *Save as Template* option in the chart's context menu writes the contents of the given chart into the `charttemplates` directory of the project.

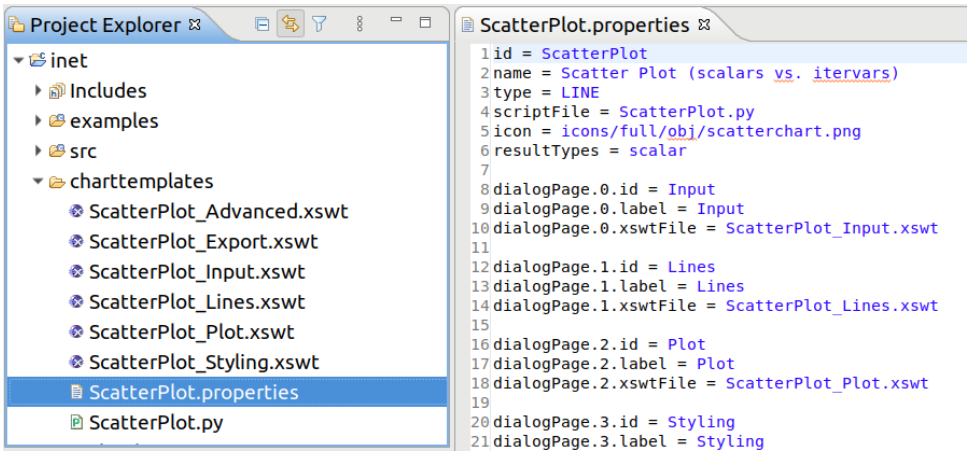


Fig. 9.18: An Exported Chart Template

You may want to tweak some properties (e.g., the descriptive name) of the saved chart template before use, but regardless, the new chart template is immediately available for use.

9.14.2 Parts of a Chart Template

A chart template consists of several parts, describing the initial contents of charts created from it: what kind of drawing widget it needs (Matplotlib or one of the native plot widgets), what script it executes, how its configuration dialog looks like, what types of result items it can process/show, and which icon should be used for it.

Namely, there are several files:

- `<name>.properties`: This is the main file. It defines the name and other attributes of the chart template and references all other files by name. The syntax is Java property file.
- `<name>.py`: The Python file that contains the chart script.
- `*.xswt`: The dialog pages.

Note: Scripts and dialog pages can be shared by multiple chart templates.

Notable keys in the properties file:

- `id`: Internal identifier
- `name`: Descriptive name
- `type`: `MATPLOTLIB`, or one of `LINE`, `BAR`, and `HISTOGRAM` for native plots
- `scriptFile`: The chart script Python file
- `icon`: Icon file, e.g. in PNG format
- `resultTypes`: One or more of `scalar`, `vector`, `parameter`, `histogram`, and `statistics`, separated by commas
- `description`: Long description of the chart in HTML format
- `dialogPage.<n>.id`: Internal identifier of the nth dialog page
- `dialogPage.<n>.label`: Label of the tab of the nth dialog page
- `dialogPage.<n>.xswtFile`: XSWT file of the nth dialog page

9.15 Under the Hood

This section details the internal workings of the Python integration in the Analysis Tool. Its contents are not directly useful for most users, only for those who are curious about the technicalities or want to troubleshoot an issue.

Chart scripts are executed by separate Python processes, launched from the `python3[.exe]` found in `$PATH`. This decision was made so that a rogue chart script can't make the entire IDE unresponsive or crash it. Also, it's possible to put resource or permission constraints on these processes without hindering the IDE itself, and they can be killed at any time with no major consequences to the rest of the Analysis Tool - for example, in the event of a deadlock or thrashing.

These processes are ephemeral, and a fresh one is used for each refresh, so no interpreter state is preserved across executions. A small number of processes are kept pre-spawned in a pool, so they can be put to use quickly when needed.

If you wish to utilize virtual environments, start the entire IDE from a shell in which the environment to use has been activated. This way, the spawned Python interpreter processes will also run in that environment.

The level of flexibility offered by this arbitrary scripting unfortunately comes with its own dangers too. Note that the scripts running in charts have full access to everything on your computer without any sandboxing, so they can read/write/delete files, open graphical windows, make network connections, utilize any hardware resources, etc.! Because of this, make sure to only ever open analysis files from sources you trust! (Or open files from untrusted sources only on systems that are not critical.)

Communication between the Eclipse IDE and the spawned Python processes is done via the Py4J project, through an ordinary network (TCP) socket.

To avoid the CPU and RAM inefficiencies caused by the string-based nature of the Py4J protocol, bulk data is transferred in shared memory (POSIX SHM or unnamed file mappings on Windows) instead of the socket. Without this, binary data would have to be base64 encoded, then represented as UTF-16, which would be about 3x the size on top of the original content, which is already present in both processes. Data passed this way includes any queried results (in pickle format), and in the other direction, the data to plot on native plot widgets, or the raw pixel data rendered by Matplotlib.

Many other kinds of information, like GUI events or smaller pieces of data (like chart properties) are passed through the Py4J socket as regular function call parameters.

CHAPTER

TEN

NED DOCUMENTATION GENERATOR

10.1 Overview

This chapter describes how to use the NED Documentation Generator from the IDE.

Please refer to the OMNEST Manual for a complete description of the documentation generation features and the available syntax in NED and MSG file comments.

The generator has several project-specific settings that can be set from the project context menu through the Properties menu item. The output folders for both NED documentation and C++ documentation can be set separately. The doxygen-specific configuration is read from the text file `doxy.cfg` by default. The IDE provides a sensible default configuration for doxygen in case you do not want to go through all the available options. The generated HTML uses CSS to make its style customizable. You can provide your own style sheet if the default does not meet your needs. In general, all project-specific settings have good defaults that work well with the documentation generator.

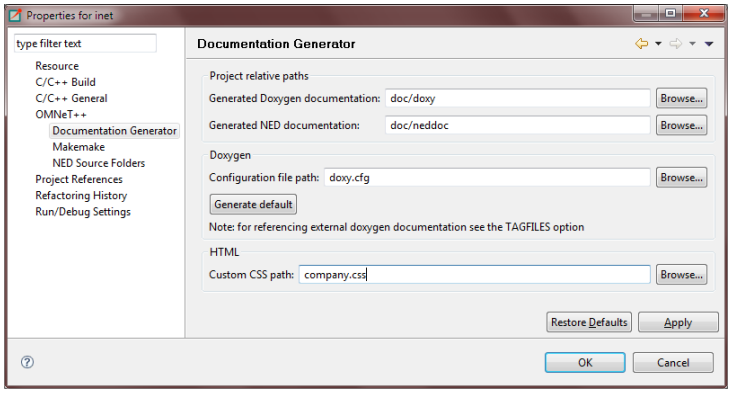


Fig. 10.1: Configuring project-specific settings

To generate NED documentation, you need to select one or more projects. Then, either go to the main *Project* menu or to the project context menu and select the *Generate NED Documentation* menu item. This will bring up the configuration dialog where you can set various settings for the current generation before starting it.

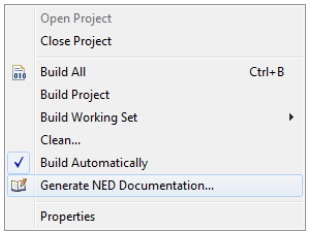


Fig. 10.2: Opening the NED documentation generator

The IDE can generate documentation for multiple projects at the same time. Other options control the content of the documentation, including what kind of diagrams will be generated and whether NED sources should be included. You can enable doxygen to generate C++ documentation that will be cross-linked from the NED documentation. The tool can generate the output into each project as configured in the project-specific settings, or into a separate directory. The latter is useful for exporting standalone documentation for several complex projects at once.

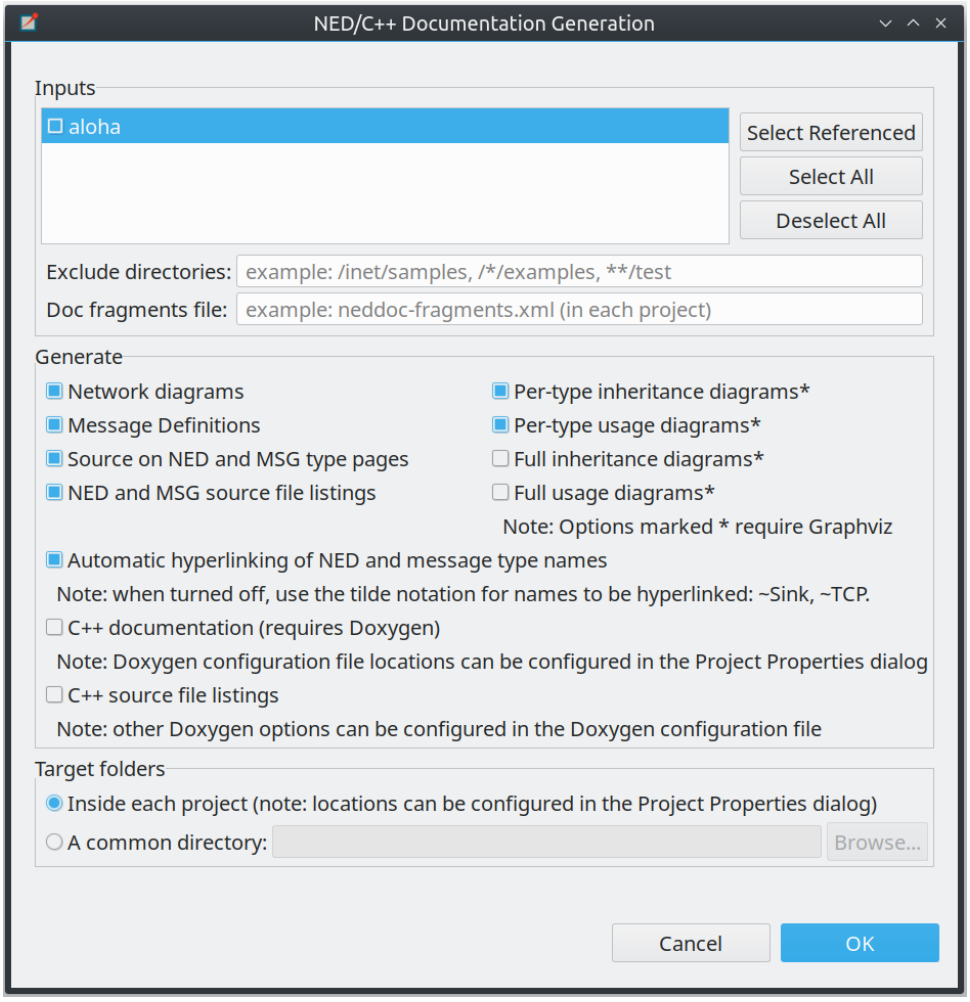


Fig. 10.3: Configuring the NED documentation generator

The NED generation process might take a while for big projects, so please be patient. For example, building the complete documentation for the INET project, including the C++ doxygen documentation, takes a few minutes. You can track the progress in the IDE's progress monitor.

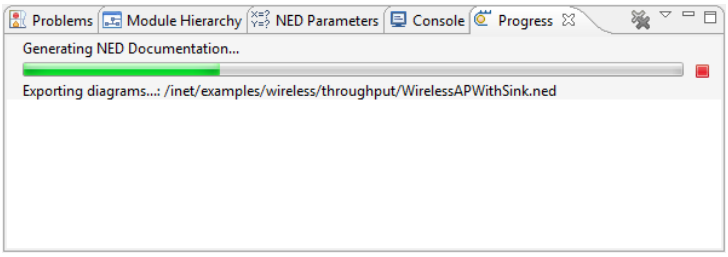


Fig. 10.4: Generating NED documentation in progress

The result is a number of cross-linked HTML pages that can be opened by double-clicking the generated `index.html`. On the left side, you will see a navigation tree, while on the right side, there will be an overview of the project. If you have not yet added a `@titlepage` directive to your NED comments, then the overview page will display default content.

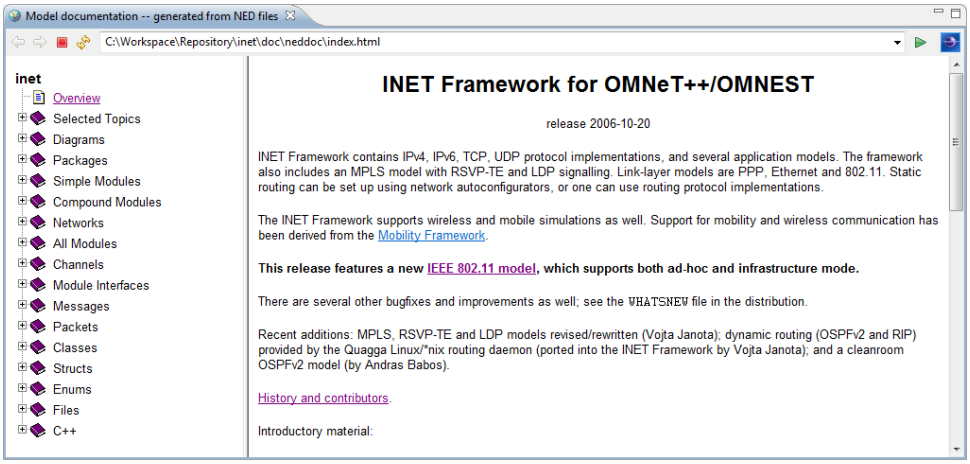


Fig. 10.5: The resulting NED documentation

The documentation contains various inheritance and usage diagrams that make it easier to understand complex models. The diagrams are also cross-linked, so when you click on a box, the corresponding model element's documentation will be opened. The NED model elements are also exported graphically from the NED Editor. These static images provide cross-referencing navigation for submodules.

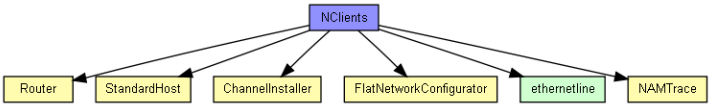


Fig. 10.6: NED usage diagram

There are also a number of tables that summarize various aspects of modules, networks, messages, packets, etc. The most interesting is the list of assignable parameters. It shows all parameters from all submodules down the hierarchy that do not have fixed values. These can be set either through inheritance, encapsulation, or from the INI file as experiments.

Assignable submodule or channel parameters:

Name	Type	Default value	Description
configurator.networkAddress	*string*	"192.168.0.0"	network part of the address (see netmask parameter)
configurator.netmask	*string*	"255.255.0.0"	host part of addresses are autoconfigured
nam.logfile	*string*	"trace.nam"	
nam.prolog	*string*	""	
r1.routingFile	*string*	""	
r1.networkLayer.proxyARP	*bool*	true	
r1.networkLayer.ip.procDelay	*double*	0s	
r1.networkLayer.arp.retryTimeout	*double*	1s	number seconds ARP waits between retries to resolve an IP address
r1.networkLayer.arp.retryCount	*int*	3	number of times ARP will attempt to resolve an IP address
r1.networkLayer.arp.cacheTimeout	*double*	120s	number seconds unused entries in the cache will time out
r1.ppp[*].queueType	*string*	"DropTailQueue"	
r1.ppp[*].ppp.mtu	*int*	4470	
r1.eth[*].queueType	*string*	"DropTailQueue"	
r1.eth[*].mac.promiscuous	*bool*	false	if true, all packets are received, otherwise only the ones with matching destination MAC address
r1.eth[*].mac.address	*string*	"auto"	MAC address as hex string (12 hex digits), or "auto". "auto" values will be replaced by a generated MAC address in init stage 0

Fig. 10.7: NED assignable parameters

There are other tables that list parameters, properties, gates, using modules or networks, and various other data along with the corresponding descriptions. In general, all text might contain cross-links to other modules, messages, classes, etc. to make navigation easier.

CHAPTER

ELEVEN

EXTENDING THE IDE

There are several ways to extend the functionality of the OMNEST IDE. The Simulation IDE is based on the Eclipse platform but extends it with new editors, views, wizards, and other functionality.

11.1 Installing New Features

Because the IDE is based on the Eclipse platform, it is possible to add additional features that are available for Eclipse. The installation procedure is exactly the same as with a standard Eclipse distribution. Choose the *Help* → *Install New Software* menu item and select an existing Update Site to work with or add a new Site (using the site URL) to the Available Software Sites. After the selection, you can browse and install the packages the site offers.

To learn about installing new software into your IDE, please visit the *Updating and installing software* topic in the *Workbench User Guide*. You can find the online help system in the *Help* → *Help Contents* menu.

Tip: There are thousands of useful components and extensions for Eclipse. The best places to start looking for extensions are the Eclipse Marketplace (http://marketplace.eclipse.org/) and the Eclipse Plugins info site (http://www.eclipse-plugins.info).

11.2 Adding New Wizards

The Simulation IDE makes it possible to contribute new wizards to the wizard dialogs under the *File* → *New* menu without writing Java code or requiring any knowledge of Eclipse internals. Wizards can create new simulation projects, new simulations, new NED files, or other files by using templates or perform export/import functions. Wizard code is placed under the `templates` folder of an OMNEST project, which makes it easy to distribute wizards with the model. When the user imports and opens a project that contains wizards, the wizards will automatically become available.

Tip: The way to create wizards is documented in the ide-customization-guide.

11.3 Project-Specific Extensions

It is possible to install an Eclipse plug-in by creating a `plugins` folder in an OMNEST project and copying the plug-in JAR file to that folder (this mechanism is implemented as part of the Simulation IDE and does not work in generic Eclipse installations or with non-OMNEST projects). This extension mechanism allows the distribution of model-specific IDE extensions together with a simulation project without requiring the end user to do extra deployment steps to install the plug-in. Plugins and wizards that are distributed with a project are automatically activated when the host project is opened.

Eclipse plug-in JAR files can be created using the Plug-in Development Environment. The OMNEST IDE does not contain the PDE by default; however, it can be easily installed if necessary.

Tip: Read the ide-developers-guide for more information on how to install the PDE and how to develop plug-in extensions for the IDE.
